VŠB TECHNICKÁ UNIVERZITA OSTRAVA
FAKULTA ELEKTROTECHNIKY A INFORMATIKY

Modul SARSA pro Modeler neuronových sítí

Module SARSA for Neuron Net Modeler

Tomáš Mariňák

Diplomová práce

Vedoucí práce: Ing. David Ježek, Ph.D.

Ostrava, 2021

Abstrakt

Táto práca sa zaoberá rozšírením programu NeuronNetModeler o ďalší nový modul, ktorý obsahuje reinforcement learning algoritmus SARSA a deep reinforcement learning algoritmus Deep SARSA network. Ďalším jej cieľom je paralelizácia učenia siete Deep SARSA na základe dátovej paralelizácie. V tejto práci sú popísané spomínané algoritmy a ich proces učenia. Ďalej práca obsahuje dátovú paralelizáciu algoritmu Deep SARSA network, ktorá je spúšťaná na distribuovaných uzloch superpočítača. S využitím týchto implementácií boli vykonané experimenty, ktoré sa zameriavali na ich efektivitu a rýchlosť učenia.

Kľúčové slová

Reinforcement learning, SARSA, Deep SARSA network, Neurónové siete, Dátová paralelizácia

Abstract

This master thesis describes extension of the program NeuronNetModeler by new module, that consists of reinforcement learning algorithm SARSA and deep reinforcement learning algorithm called Deep SARSA network. Its next goal is parallelization of Deep SARSA learning based on data parallelization. In this thesis are described already mentioned algorithms and their process of learning. Furthermore this thesis contains data parallelization of Deep SARSA network algorithm, that is runned on distributed nodes of supercomputer. Experiments were done with the usage of this implementation, which were focused on their efficiency and speed of learning process.

Keywords

Reinforcement learning, SARSA, Deep SARSA network, Neural networks, Data parallelization

Poďakovanie

Rád by som na tomto mieste poďakoval vedúcemu práce Ing. Davidovi Ježkovi Ph.D za všetky rady a nápady počas vypracovania tejto práce. Ďalej by som chcel poďakovať všetkým, ktorí mi pomohli počas doby celého môjho štúdia.

Obsah

Zoznam použitých skratiek a symbolov

ANN	–	Artificial Neural Network
RL	–	Reinforcement Learning
DSN	–	Deep SARSA network
DQN	–	Deep Q network
MDP	–	Markov Decision Process
MSE	–	Mean squared error
ERM	–	Experience Replay Memory

Zoznam obrázkov

Zoznam tabuliek

Kapitola 1

Úvod

1.1 Popis problematiky

Táto práca sa zaoberá problematikou, ktorá už vo svete nie je nová ale je často používaná, kombinácia umelých neurónových sietí a reinforcement learning algoritmu je popísaná v literatúre veľakrát. Avšak algoritmus, ktorý je väčšinou použitý pre učenie spomínaných sietí je jednoduchý inkrementálny algoritmus. Pre učenie sa tiež môžu využiť aj on-line algoritmy a to SARSA alebo Q-learning. V ďalších kapitolách sa popisuje ako samotný algoritmus SARSA, taktiež neurónové siete a následne ich kombinácia. Následne práca pokračuje riešením problému dátovej paralelizácie učenia siete. Táto paralelizácia je implementovaná aj na superpočítač s využitím knižnice AERON ako na jeden z jeho uzlov tak aj väčší počet. Na záver sú zdokumentované experimenty s rôznymi typmi implementácie algoritmu SARSA na viacerých prostrediach a taktiež v sekvenčnom a paralélnom behu.

1.2 Motivácia

V mojej semestrálnej práci som sa zaoberal algoritmom SARSA, následne pri volení témy pre diplomovú prácu som sa dopočul o tzv.*Deep Q-Learning*, ktorý využíva kombináciu reinforcement learning algoritmu a umelej neurónovej siete, keďže algoritmus SARSA taktiež spadá do oblasti *Reinforcement learning* zaujímalo ma, či je možné tento algoritmus využiť taktiež v kombinácii s neurónovou sieťou. Najčastejšie zvoleným algoritmom pri kombinácii umelých neurónových sietí a *Deep Learning* je teda off-policy algoritmus Q-learning. Táto kombinácia sa nazýva v skratke DQN, menej známou je Deep SARSA network, kvôli čomu som sa rozhodol ukázať, že aj táto možnosť dokáže riešiť dané problémy a dokáže sa vyrovnať DQN.

1.3 Ciele

Práca má 4 hlavné oblasti:

- Oboznámenie sa s problematikou Deep Reinforcement Learning algoritmov.
- Implementácia algoritmu SARSA spoločne s neurónovou sieťou.
- Implementácia paralelizácie algoritmu Deep SARSA network.
- Experimenty nad rôznymi testovacími prostrediami.

Táto sekcia popisuje rozsah spomínaných oblastí a ich jednotlivých cieľov. Tieto ciele dohromady tvoria hlavný cieľ pre diplomovú prácu.

1.3.0.1 Oboznámenie sa s problematikou Deep Reinforcement Learning algoritmov

V tejto sekcii sa popisujú jednotlivé technológie a termíny potrebné pre pochopenie fungovania Deep Reinforcement Learning algoritmov, presnejšie algoritmu Deep SARSA network. Sekcia začína popisom strojového učenia, reinforcement learning až deep reinforcement learning.

1.3.0.2 Implementácia algoritmu SARSA spoločne s neurónovou sieťou

Táto implementácia spočiatku zahrňuje popis vývoja samotného algoritmu SARSA a jeho modifikácií, následne sa zameriava na implementáciu neurónovej siete spoločne so spomínaným algoritmom. Ako ďalšie popisuje implementáciu modifikácií potrebných pre zvýšenie stability a efektivity Deep SARSA network ako sú Experience Memory Replay a Target Network.

1.3.0.3 Implementácia paralelizácie algoritmu Deep SARSA network

Táto sekcia zahŕňa implementáciu dátovej paralelizácie algoritmu Deep SARSA network, ktorej účelom bolo vytvoriť algoritmus, ktorý bude schopný riešiť dané prostredie vďaka paralélnemu behu. Tento beh bol reprezentovaný pomocou daných architektúr, ktoré popisujú problematiku dátovej paralelizácie.

1.3.0.4 Experimenty nad rôznymi testovacími prostrediami

Poslednou oblasťou tejto práce sú experimenty nad samotným algoritmom SARSA, následne Deep SARSA network a viacerými testovacími prostrediami ako je Lunar landing alebo Cart pole balancing. Tieto experimenty sa zameriavajú na porovnanie výsledkov rôznych kombinácií modifikácií, ktoré obsahuje Deep SARSA network alebo iba algoritmu SARSA a taktiež kombinácií hyperparametrov. Poslednými experimentami sú pokusy dátovej paralelizácie na superpočítači od spoločnosti IT4innovations. V závere sú zhrnuté výsledky týchto experimentov ale aj celkové zhrnutie celej diplomovej práce.

Kapitola 2

Strojové učenie

Je smer zaoberajúci sa počítačovými algoritmami, ktoré využívajú skúsenosti pre ich automatické vylepšovanie, bez toho aby boli na to explicitne naprogramované. Tieto programy sú naprogramované pre prístup k dátam, kvôli tomu aby ich mohli využiť pre učenie samých seba [1] . Strojové učenie spadá do širšej oblasti zvanej umelá inteligencia a delí sa na tri hlavné pod-oblasti:

- učenie pod dohľadom;
- učenie bez dohľadu;
- reinforcement learning;

2.1 Učenie pod dohľadom

Využíva znalosti získané z minulosti pre predpoveď budúcich udalostí. Tieto znalosti sú reprezentované pomocou učenia na tzv.trénovacej množine. Trénovacia množina obsahuje vstupy pre algoritmus, kde každý vstup obsahuje dáta aj o jeho očakávanom výstupe. Vďaka týmto poskytovaným dátam je algoritmus schopný naučiť sa predpoveď výstupu pre neznáme vstupy z tej istej oblasti ako sú vstupy zo spomínanej množiny[1].

2.2 Učenie bez dohľadu

Na rozdiel od učenia pod dohľadom, táto pod-oblasť nepracuje s dátami, ktoré sú "oštítkované" alebo "klasifikované". Takéto algoritmy nedokážu zistiť správny výstup, ale skúmajú dáta medzi ktorými dokážu nájsť spojitosti a následne odhaliť skryté štruktúry v týchto dátach[1].

2.3 Reinforcement learning

Odlišuje sa od učenia pod dohľadom, tým, že nevyužíva "oštítkované" vstupno/výstupné páry (labeled pairs). Zameriava sa na nájdenie rovnováhy medzi prieskumom (exploration) neznámeho prostredia a využívaním súčasných poznatkov(exploitation).

Prostredie použité pre reinforcement learning je typicky uvádzané vo forme Markovho rozhodovacieho procesu. Táto práca sa bližšie zameriava na túto problematiku.

Kapitola 3

Reinforcement learning

Jedná sa o oblasť, ktorá využíva pre učenie tzv. agenta, agentovou úlohou je vykonávať akcie v danom prostredí s cieľom maximalizovať možnú kumulatívnu odmenu získanú z daného prostredia. Takéto prostredie ja najčastejšie uvádzané vo forme Markovho Rozhodovacieho Procesu (MDP).

3.1 Algoritmy

Pod algoritmy, ktorých prostredia sú definované pomocou MDP spadajú dve kategórie algoritmov:

- založené na modeli (model-based);
- nezaložené na modeli (model-free);

3.1.0.1 Založené na modeli

Používajú model prostredia, t.j. funkciu, ktorá predpovedá prechody stavu a odmien. Model môže byť odovzdaný agentovi alebo môže byť naučený vďaka nemu. Využívanie modelu umožňuje agentovi "myslieť dopredu".

3.1.0.2 Nezaložené na modeli

Tieto algoritmy nezískavajú ani nevytvárajú spomínaný model, nedokážu predpovedať odmenu stavu bez toho aby vykonali na ňom nejakú akciu.

Medzi tieto algoritmy patria: Monte Carlo, Q-learning, SARSA.

3.2 Základné elementy

Algoritmy nezaložené na modeli využívajú mnoho elementov, ktoré je potrebné popísať vzhľadom k chápaniu celej problematiky.

3.2.0.1 Agent

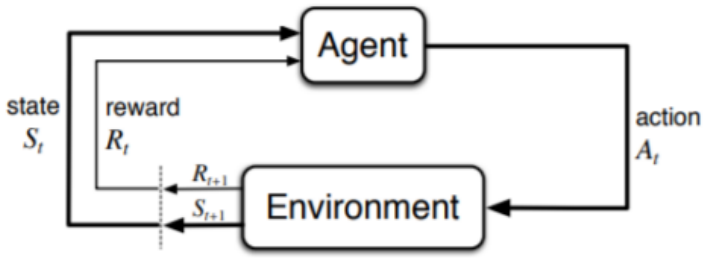
Časť algoritmu, ktorej úlohou je vykonávať rozhodnutia a na základe nich aj akcie na danom prostredí. Agenta môžeme vnímať ako prepojenie medzi prostredím a učiacim algoritmom. Toto prepojenie sa rozhoduje akú akciu zvolí na základe učiaceho algoritmu, ktorému poskytuje dáta na učenie z daného prostredia.

3.2.0.2 Markovov rozhodovací proces

Pomenovaný po ruskom matematikovi Andrejovi Markovovi. Je to diskrétny, stochastický a kontrolovaný proces v matematike. Ponúka riešenie pre modelovanie konania rozhodnutí v situáciach kde sú výsledky z časti náhodné a z časti pod kontrolou konateľa rozhodnutí(agent).

V každom časovom okamihu je proces v určitom stave *s* a konateľ môže zvoliť akúkoľvek akciu *a*, ktorá je dostupná pre daný stav. Markovov rozhodovací proces na túto akciu v nasledujúcom časovom okamihu reaguje náhodným presunutím do nového stavu *s'* a dáva konateľovi odmenu *Ra(s, s')*.

Pravdepodobnosť, že Markovov rozhodovací proces zvolí *s'* ako nový stav, je ovplyvnená zvolenou akciou. Táto pravdepodobnosť je určená funkciou prechodu *Pa(s, s')*. To znamená, že nasledujúci stav *s'* závisí na stave *s* a na užívateľovej akcii *a* [2].



Obr. 3.1: Typické prostredie riešené Markovovým rozhodovacím procesom

3.2.0.3 Ohodnocovacia funkcia

Je to funkcia značená ako *V(s)*, ktorá hodnotí ako výhodné je byť v danom špecifickom stave *s*(alebo ako výhodné je vykonať danú akciu v danom stave s). Výhodnosť stavu ohodnocuje pomocou odmien, ktoré sú získavané.

3.2.0.4 Politika (Policy)

Politika označená ako π, je mapovanie z daného stavu *s* do pravdepodobností zvolenia každej možnej akcie z daného stavu *s*.

Inak povedané, politika popisuje správanie agenta v reinforcement learning algoritme. Je to funkcia, ktorá pre daný stav a akciu vráti pravdepodobnosť vykonania akcie v danom stave.

3.2.0.5 Bellmanova rovnica

Pomenovaná podľa amerického matematika Richarda Bellmana. V reinforcement learning algoritme je využívaná pre riešenie Markovho rozhodovacieho procesu [3]. V stochastických problémoch sa využíva pre optimálnu politiku v podobe:

$$V(s) = max_a(R(s,a) + \gamma \sum_{n=s'} P(s,a,s')V(s')), \text{ kde:}$$

- **R(s,a)**je odmena pre stav s a akciu a vykonanú z tohto stavu;
- **discount factor** je $\gamma \in [0,1]$;
- $\sum_{n=s'} P(s,a,s')V(s')$ je suma pravdepodobností zvolenia jednotlivých možných stavov;
- max_a je ohodnocovacia funkcia pre výber stavu s s maximálnou hodnotou;

3.3 Algoritmus SARSA

Je to On-Policy algoritmus, využívajúci Markovov rozhodovací proces. Jeho meno odrzkadľuje jeho hlavnú funkciu pre učenie, resp. aktualizovanie Q-hodnoty pre pár stavu a akcie $\mathbf{Q}(S_t,A_t)$, na základe:

- aktuálneho stavu S_t;
- akcie A_t zvolenej agentom z aktuálneho stavu;
- odmeny R_t, ktorú agent získa z prostredie na základe zvolenej akcie A_t;
- nasledujúceho stavu S_{t+1} do ktorého sa agent dostane po vykonaní akcie A_t;
- akcie A_{t+1}, ktorú si agent zvolí v novom stave;

Akronym pre päticu $(S_t, A_t, R_t, S_{t+1}, A_{t+1})$ je teda SARSA [4].

3.4 On policy vs Off policy

Každý algoritmus v reinfocement learning, ktorý nie je založený na modeli musí nasledovať nejakú politiku aby bol schopný rozhodovať aké akcie vykoná pre každý stav. Napriek tomu, samotné učenie algoritmu nemusí využívať žiadnu politiku. Algoritmy, ktoré sa týkajú politiky, ktorá je využívaná aj pre učenie sú označované ako On-Policy algoritmy(SARSA). Tie, ktoré nevyužívajú žiadnu politiku pre učenie sú označované ako Off-policy algoritmy(Q-Learning).

3.5 Popis algoritmu

- Inicializácia Q-tabuľky, ktorá bude obsahovať všetky páry stav-akcia a ich Q-hodnotu
- Následne pre každú epizódu:
  - Inicializácia počiatočného stavu **s**, buď sa jedná o náhodný stav alebo konkrétny(záleží na prostredí)
  - Zvoľ akciu **a** pre stav **s**, na základe zvolenej politiky založenej na Q-hodnote
  - Opakuj, kým **s** nie je konečný stav:
    - Vykonaj akciu **a** a zisti odmenu a ďalší stav **s'**
    - Zvoľ akciu **a'** pre stav **s'**, na základe zvolenej politiky založenej na Q-hodnote
    - Aktualizuj Q-hodnotu pre **Q(s,a)** na základe vzorca:
      $Q(s,a) = Q(s,a) + \alpha(r + \gamma Q(s',a') + Q(s,a))$
    - Do aktuálneho stavu **s** vlož stav **s'**
    - Do aktuálnej akcie **a** vlož akciu **a'**

3.6 Modifikácie algoritmu

Tento algoritmus má rôzne varianty, ktoré sa môžu líšiť od jeho základu, medzie tieto varianty patria napríklad:

- Eligibility Traces - verzia obohatená o problematiku Eligibility Traces, pre ohodnocovanie celkovej(čiastočnej) cesty/postupu algoritmu, nie iba jeho najaktuálnejšieho kroku. Konkrétne si metódu *Eligibility traces* môžeme predstaviť ako reprezentáciu záznamu dočasnej udalosti, ako napríklad návšteva stavu alebo vykonanie akcie.
- Q-Sarsa(λ) - Kombinácia off a on policy metód vrátane problematiky Eligibility traces, využíva hyperparameter *omicron*, ktorý definuje pomer vplyvu jednotlivých metód. Najčastejšie ide o kombináciu učení algoritmu SARSA a Q-Learning.

Kapitola 4

Umelé neurónové siete

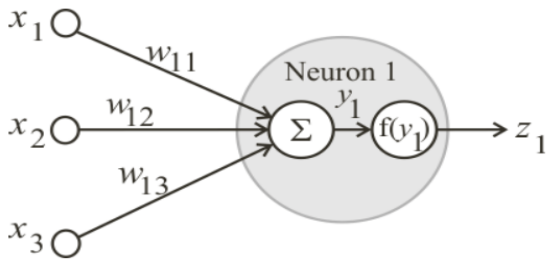
Reprezentujú výpočetné štruktúry inšpirované biologickými neurónovými sieťami, ktoré tvoria biologický mozog. Základným prvkom je neurón, ktorý je prepojený s množstvom ďalších neurónov pomocou prepojení. Spojenia biologického neurónu sú modelované v umelej neurónovej sieti ako váhy. Váhy s pozitívnou hodnotou odrážajú excitačné(budivé) spojenie, zatiaľ čo záporné hodnoty znamenajú inhibičné(tlmivé) spojenia. Všetky vstupy sú upravené váhou a sčítané. Táto aktivita sa označuje ako lineárna kombinácia. Nakoniec aktivačná funkcia riadi amplitúdu výstupu [5].

4.1 Základné elementy

4.1.0.1 Neurón

Je základnou jednotkou umelej neurónovej siete, jeho štruktúra je inšpirovaná biologickým neurónom(obrázok 4.1). Každý umelý neurón obsahuje vstupy a produkuje výstup, ktorý môže byť zaslaný do ďalších neurónov. Vstup môžu reprezentovať konkrétne dáta(obrázky, text, vektory číslíc atď.) na ktorých sa má sieť učiť alebo to môžu byť výstupy iných neurónov.

Pre získanie výstupu neurónu, ako prvé spočítame sumu vstupov vynásobených váhami(aktivácia) a pridáme bias. Táto vážená suma je poslaná do aktivačnej funkcie pre vytvorenie výstupu.



Obr. 4.1: Štruktúra neurónu

4.1.0.2 Váhy a spojenia

Sieť pozostáva zo spojení, každé spojenie poskytuje výstup jedného neurónu ako vstup do iného neurónu. Všetky spojenia majú pridelenú váhu, ktorá reprezentuje ich relatívnu významnosť.

4.1.0.3 Bias

Bias umožňuje posunúť aktivačnú funkciu vďaka pridaniu konštanty(given bias) do vstupu.

4.1.0.4 Propagačné funkcie

Propagačné funkcie majú za úlohu vypočítať vstup do neurónu z výstupov jeho predošlých neurónov a ich spojení ako váženú sumu. Termín bias môže byť pridaný do výsledku propagácie.

4.2 Spôsob učenia

Učenie umelej neurónovej siete spočíva v adaptácií samej seba voči lepšiemu spracovaniu úlohy ktorá obsahuje vzorové dáta. Učenie zahŕňa upravovanie spomínaných váh, ktorého cieľom je zvýšiť presnosť výsledku. Tento efekt docielime vďaka minimalizovaniu chýb, vzniknutých medzi výstupom siete a hodnotou skutočnej trénovacej vzorky. Učenie sa považuje za úspešne ukončené akonáhle spracovanie nových testovacích vzoriek užitočne neznižuje chybovosť.

4.2.0.1 Forward propagation

Je prvou fázou učenia, ktorá nastáva keď je sieť vystavená trenovacím vzorkám a tieto vzorky prechádzajú postupne celou neurónovou sieťou aby sa vypočítala ich predpoveď(prediction). To znamená, že vstupné dáta sú posielané cez sieť v zmysle, že všetky jej neuróny aplikujú ich transformáciu do informácie, ktorú získali z neurónov z predošlej vrstvy a posielajú ju neurónom z ďalšej vrstvy. Keď dáta prejdú všetkými vrstvami siete a všetky jej neuróny vykonali svoje výpočty, pristúpi sa ku výstupnej vrstve s výsledkom predpovede pre vzorkové vstupné dáta.

4.2.0.2 Loss function

Ďalšou fázou v procese učenia je využitie tzv.*Loss function*, ktorej úlohou je ohodnotiť stratu(loss) alebo chybu(error) a porovnať a zmerať ako dobrá/zlá bola predpoveď vo vzťahu ku správnemu výsledku(label). Ideálnou stratou alebo chybou je hodnota *0*, to znamená aby nebol žiadny rozdiel medzi predpoveďou a očakávanou hodnotou. Kvôli tomu, behom času, ako je model siete trénovaný, váhy spojení neurónov sú postupne upravované pokým nie sú získavané dobré predpovede.

4.2.0.3 Back propagation

Akonáhle je strata/chyba vypočítaná, táto informácia je propagovaná späť. Začínajúc výstupnou vrstvou, ktorá informáciu straty posiela do všetkých neurónov vrstvy siete, ktoré prispeli priamo k výsledku výstupu. Avšak tieto neuróny nezískajú úplný signál straty, ale iba jeho frakciu na základe ako relatívne prispeli k výstupu. Tento proces je opakovaný pre každú vrstvu až pokým všetky neuróny v sieti nezískali signál straty, ktorý popisuje ich relatívny príspevok k absolútnej strate.

Kapitola 5

Deep learning

Patrí do širšej rodiny zvanej strojové učenie, systémy tejto pod-oblasti sú založené na umelých neurónových sieťach, ktoré sú schopné učiť sa s/bez dohľadu a z dát, ktoré môžu byť ale aj nemusia byť štrukturované alebo označené(labeled).

Prívlastok "deep"(hlboký) odkazuje na používanie väčšieho počtu skrytých vrstiev v sieti, klasické umelé neurónové siete majú malý počet skrytých vrstiev, ktorý sa pohybuje okolo čísla 3, pokým siete patriace do deep learning ich môžu mať 150. Modely deep learning sú zväčša trénované pomocou veľkej trénovacej množiny, ktorej dáta sú označené(labeled) a architektúry umelej neurónovej siete, ktorá sa učí vlastnosti priamo z dát bez akéhokoľvek manuálneho prístupu pre extrakciu týchto vlastností [6].

5.1 Typy Deep learning sietí

Existuje viacerov typov Deep learning sietí, ktoré sa využívajú vo viacerých odvetviach priemyslu.

5.1.0.1 Deep reinforcement learning

Je to pod-oblasť, ktorá kombinuje deep learning a reinforcement learning. Reinforcement learning využíva agenta, ktorý vykonáva akcie spočiatku na princípe pokus-omyl, následne na základe najvhodnejšej akcie pre daný stav. Deep reinforcement learning pridáva do riešenia deep learning sieť, čo znamená, že umožňuje agentom vykonávať rozhodnutia z neštrukturovaných vstupných dát bez manuálneho spracovania stavového priestoru. Tieto siete sú schopné prijímať veľké vstupy(napr. každý pixel obrazovky vo video hre) a rozhodnúť sa akú akciu vykonať pre optimalizovanie cieľa(maximalizovanie skóre v hre) [7].

5.1.0.2 Konvolučné siete

Sú to siete, ktoré sa využívajú najmä pre prácu s grafickými dátami, ako sú obrázky či video. Tieto siete sú regularizované verzie viac-vrstvých perceptronov, čo znamená, že ich neuróny v jednej sietí sú prepojené so všetkými neurónmi v dalšej sieti. Ich základným blokom sú konvolučné vrstvy, ktoré využívajú tzv. filter pre spracovanie časti vstupu.

5.1.0.3 Rekurentné siete

Využívajú sekvenčné dáta alebo dáta s časovým postupom. Často sa využívajú pre ordinálne alebo časové problémy ako napríklad: jazykový preklad, rozpoznávanie reči, rozpoznávanie sentimentu z textu atď. Od ostatných štruktúr sa rozlišujú svojou "pamäťou", kde získavajú informáciu z predošlého vstupu pre ovplyvnenie aktuálneho vstupu a výstupu. Výstup tejto siete je závislý na predošlých elementoch v danej sekvencii [8].

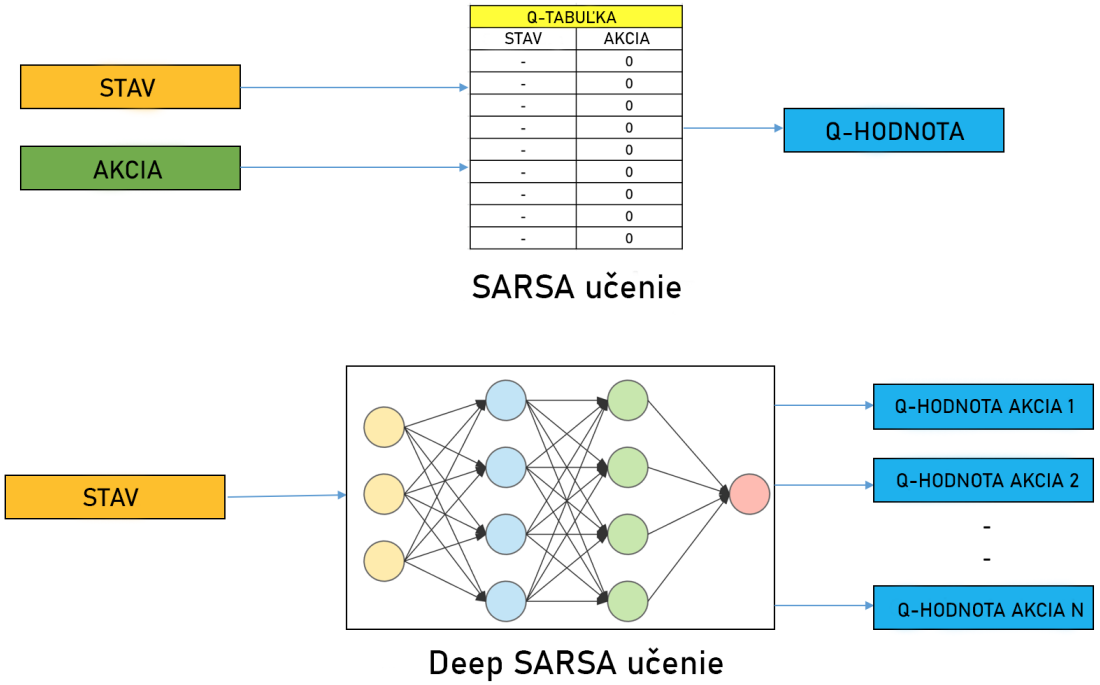
5.1.0.4 Deep belief siete

Jedná sa o siete, ktoré využívajú pravdepodobnosti a učenie bez dohľadu pre vypočítanie výstupu. Skladajú sa z binárnych latentných premenných a obsahujú neusmernené vrstvy ako aj nasmerované vrstvy. Na rozdiel od iných modelov sa každá vrstva v Deep belief sieťach učí celý vstup [9].

5.2 Deep SARSA network

Je verziou systémov patriacich do pod-oblasti Deep reinforcement learning. Tvorí kombináciu umelej neurónovej siete spoločne s Deep Learning a algoritmu SARSA, ktorý reprezentuje Reinforcement Learning. Deep SARSA network prináša riešenie pre prostredia/problémy, ktoré klasický algoritmus SARSA nebol schopný vyriešiť vzhľadom k ich komplexnosti, resp. často kvôli veľkému počtu stavov a akcii prostredia. Q-tabuľka algoritmu SARSA, ktorá obsahuje páry stav-akcia a k nim Q-hodnotu začne byť po istom vyššom počte týchto elementov neefektívna. Prístup k týmto elementom začne byť výpočetne náročný a samotný algoritmus začne byť veľmi pomalý.

Deep SARSA network tento problém rieši náhradou Q-tabuľky za umelú neurónovú sieť s pomocou využitia Deep Learning. Kde úlohou umelej neurónovej siete je taktiež aproximovať Q-funkciu. Stavy prostredia sú posielané do siete ako vstupy a ako výstupy sieť predpovedá akcie vzhľadom k vstupu. Toto riešenie je možné pochopiť na nasledujúcom obrázku 5.1:



Obr. 5.1: Porovnanie učenia SARSA a Deep SARSA network [10].

5.3 Problémy pri Deep SARSA network

5.3.0.1 Nasledovanie nestálej cieľovej hodnoty

V klasickom Deep Learning sa cieľová hodnota pre daný vstup nemení, vzhľadom na to, že trénovacia množina sa taktiež nemení. Deep SARSA network sa líši od klasického Deep learning tým, že sa cieľová hodnota pre daný vstup mení, kvôli tomu, že táto hodnota je získavaná pomocou politiky. Tým vzniká problém, že sa Deep SARSA network učí na zakláde nestálej cieľovej hodnoty, čo vytvára nestabilitu pri učení. Riešením je pridanie ďalšej siete tzv. **Target network**, ktorá sa využíva pri učení, teda pre určovanie cieľovej hodnoty.

5.3.0.2 Náhrada Q-tabuľky

V klasickom SARSA učení algoritmus poznal Q-hodnotu pre každý preskúmaný stav prostredia, ktorému na základe učenia prideľoval Q-hodnotu. V Deep SARSA network nastáva problém, že sa sieť učí iba na aktuálnom vstupe, ktorý reprezentuje aktuálny stav prostredia. Sieť pomalšie konverguje ku správnemu riešeniu, kvôli tomu, že svoje parametre upravuje vždy iba na základe jedného vstupu. Pri prostrediach, ktoré majú veľký počet stavov môže vznikať problém s konver-

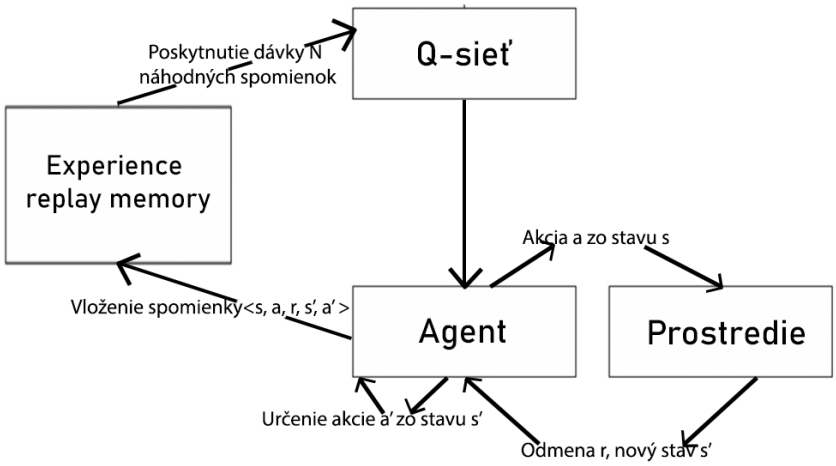
genciou. Riešenie poskytuje tzv. **Experience replay memory**, ktorá obsahuje skúsenosti získané z prostredia pomocou agenta.

5.4 Vylepšenia Deep SARSA network

5.4.0.1 Experience replay memory

Toto vylepšenie ukladá do svojej pamäte skúsenosti, získané každým krokom agenta v prostredí. Táto pamäť je nastavená na určitú veľkosť(pomocou hyperparametru), akonáhle počet skúseností v pamäti dosiahne veľkosť udanú hyperparametrom, nové skúsenosti začínajú nahradzovať najstaršie spomienky v pamäti[11].

Pre trénovanie siete sa z tejto pamäte zvolí N náhodných skúseností, počet týchto skúseností je udávaný pomocou hyperparametru. Hlavným dôvodom využitia experience memory replay v Deep SARSA network je porušenie korelácie medzi za sebou nasledujúcimi vstupmi siete.



Obr. 5.2: Využitie ERM v Deep SARSA network

Ak by sa sieť učila iba zo vstupov, ktoré majú náväznosť v prostredí, jej schopnosť naučiť sa riešiť stochastické prostredie je výrazne znížená. Keďže by sa učila iba na vstupoch, ktoré korelujú medzi sebou, vzhľadom na to že prichádzajú sekvenčne z prostredia. Táto sieť by nevedela správne predpovedať výstup pre náhodné vstupy v stochastickom prostredí. Tento problém je riešený učením na dávke náhodných skúseností, ktoré prerušujú možnú koreláciu.

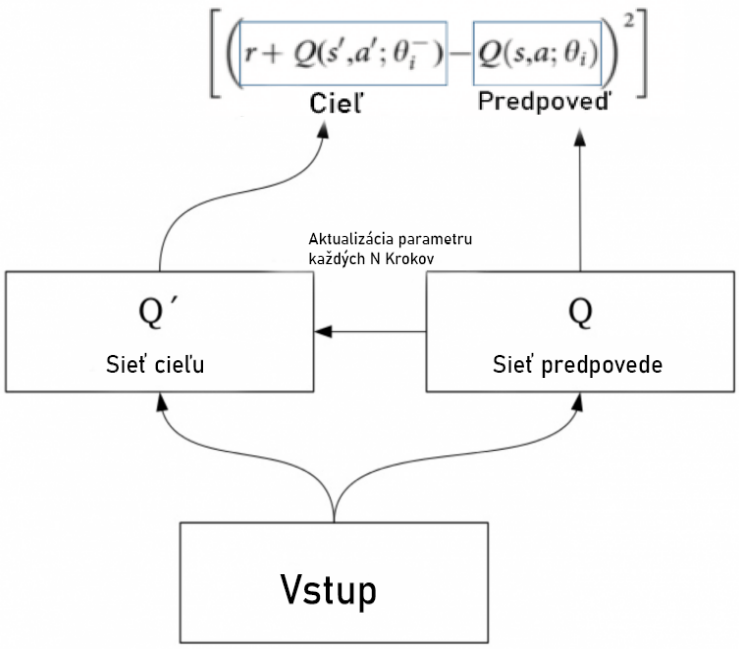
Experience Skúsenosť je popísaná ako pätica, resp. v Deep SARSA network šestica:

$$e_t = (s_t, a_t, r_t, s'_{t+1}, a'_{t+1})$$

Táto šestica reprezentuje stav prostredia s_t, akciu a_t vykonanú z tohto stavu, odmenu r_t získanú agentom v čase t ako výsledok predošlého páru stavu a akcie, nasledujúci stav prostredia s_{t+1} a akciu a_{t+1} vykonanú na základe politiky z nasledujúceho stavu. V prípade, že prostredie obsahuje terminálne stavy pridáva sa do tejto n-tice aj atribút *done*, ktorý reprezentuje, či je daný stav terminálny.

5.4.0.2 Target network

Vzhľadom na to, že tá istá sieť určuje hodnotu predpovede ako aj cieľu, môže vznikať divergencia medzi týmito dvoma hodnotami. Namiesto používania jednej umelej neurónovej siete pre učenie, je možné využiť ďalšiu sieť [11].



Obr. 5.3: Využitie Target Network v Deep SARSA network

Táto ďalšia separátna umelá neurónová sieť sa využíva pre výpočet cieľovej hodnoty. Má presne tú istú štruktúru ako má pôvodná sieť s jediným rozdielom, jej parametre sa nemenia počas učenia. Tieto parametre sú nahradzované parametrami z umelej neurónovej siete, ktorá slúži pre predpoveď(obrázok 5.3). Táto náhrada prebieha každých N krokov/iterácii (počet krokov je udávaný ako hyperparameter). Toto vylepšenie smeruje k stabilnejšiemu trénovaniu siete, pretože na určitý počet iterácii udržuje cieľovú funkciu stálu.

5.4.0.3 Popis algoritmu

V tejto sekcii je popísaný pseudokód algoritmu Deep SARSA network. V tomto algoritme sa využívajú obe spomínané modifikácie *Target network* a *Experience memory replay*.

Algoritmus 1 Deep SARSA network

Inicializácia siete Q s náhodnými váhami θ
Inicializácia target network Q^- s váhami $\theta^- = \theta$
Inicializácia experience replay memory D
Inicializácia Agenta pre interagovanie s prostredím
while pocet_epizod < max _pocet _epizod **do**
    ϵ = nastavenie novej hodnoty na základe epsilon-decay
    Zvolenie akcie a zo stavu s s použitím politiky ϵ
    Agent vykoná akciu a, získa odmenu t a nasledujúci stav **s'**
    Zvolenie akcie *a'* zo stavu *s'* s použitím politiky ϵ
    Vloženie spomienky (*s, a, r, s', a', done*) do ERM D
    if D má dostatok skúseností **then**
        Získanie dávky M náhodných N skúseností z D
        for každá skúsenosť (*s, a, r, s', a', done*) **in** M **do**
            if $done_i$ **then**
                $y_i = r_i$
            else
                $y_i = r_i + \gamma Q(s'_i, a')$
            end if
        end for
        Výpočet straty $L = 1/N \sum_{N-1}^{i=0} (Q(s_i, a_i) - y_i)^2$
        Aktualizácia Q, pomocou SGD algoritmu, pomocou minimalizovania straty L
        Každých C krokov, skopíruj váhy z Q do Q^-
    end if
end while

Kapitola 6

Implementácia

Táto sekcia diplomovej práce popisuje implementáciu Deep SARSA network v programovacom jazyku *Java 8* s využitím knižnice pre implementáciu deep learning sietí *deeplearning4j*, následne implementáciu paralelizácie učenia Deep SARSA network s pomocou komunikačnej knižnice *Aeron*.

6.1 Implementácia SARSA učenia

Implementácia učenia SARSA a algoritmu SARSA samotného pozostávalo z nižšie popísaných hlavných tried, ktoré reprezentovali celkový proces učenia(štruktúra je znázornená na obrázku 6.1).

6.1.0.1 Trieda SarsaAlgorithm

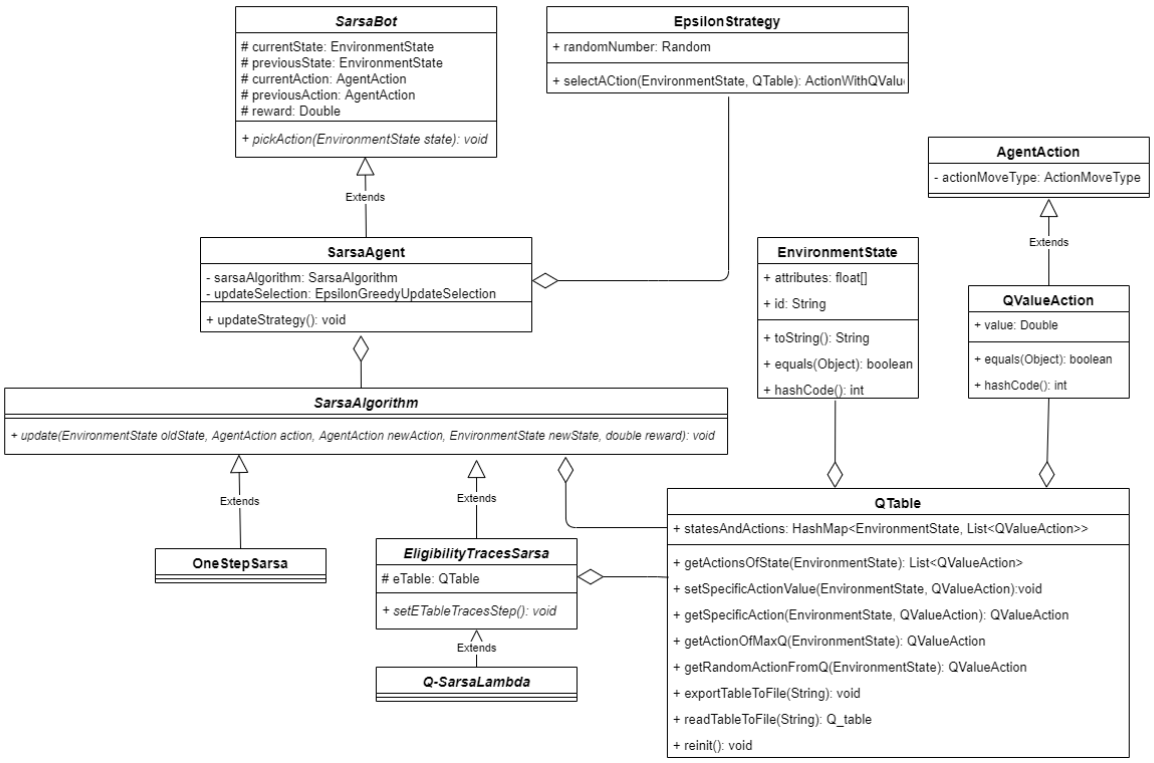
Táto trieda bola implementovaná ako abstraktná vzhľadom k tomu, že som využil viaceré modifikácie učenia. Obsahovala jedinú metódu `update` slúžiacu pre aktuálizáciu Q-hodnoty pre aktuálny stav. Využívala taktiež triedu `QTable`, ktorá obsahuje, ako z názvu tejto triedy vyplýva, celú Q-tabuľku párov stav-akcia a ich Q-hodnoty.

Túto triedu rozširovali triedy `OneStepSarsa` a `EligibilityTracesSarsa`.

6.1.0.2 Trieda QTable

Q-tabuľka potrebovala vlastnú triedu vzhľadom k tomu, že sa pomocou nej vykonávalo veľa operácií, keďže bola jediným zdrojom párov stav-akcia a ich Q-hodnoty. Tieto páry boli reprezentované pomocou triedou dátovej štruktúry `HashMap`, kde kľúčom bol samotný stav `EnvironmentState` a hodnotou bola dátová štruktúra `List` možných akcii `QValueAction`, ktoré obsahovali Q-hodnotu.

V tejto triede som taktiež implementoval možnosť serializovať jej dáta. Pomocou metódy s názvom `exportTableToFile` bolo možné po zadaní reťazcového parametra, ktorý reprezentoval názov súboru exportovať q-tabuľku do súboru. Naopak metóda `readTableFromFile` dokázala inicializovať q-tabuľku zo súboru.



Obr. 6.1: Triedny diagram pre SARSA učenie

6.1.0.3 Trieda EnvironmentState

Obsahovala pole dátových typov `float`, ktoré reprezentovali jednotlivé atribúty stavu prostredia. Tým, že stav bol naimplementovaný ako pole dátového typu `float`, bolo možné využiť túto triedu pre akékoľvek prostredia, kde atribúty stavu boli číselné hodnoty.

6.1.0.4 Trieda QValueAction

Trieda, ktorá slúžila ako reprezentácia možného typu akcie pre dané prostredie a taktiež držala Q-hodnotu pre daný typ akcie.

Rozširovala triedu `AgentAction`, kde táto trieda držala rozhranie `ActionMoveType`, ktoré slúžilo iba ako typ akcie, keďže napríklad pre Deep SARSA network nie je potrebná Q-hodnota.

6.1.0.5 Trieda EpsilonStrategy

Pre implementovanie politiky učenia som vytvoril triedu `EpsilonStrategy`, trieda využívala triedu `Random` ako vlastnosť pre generovanie náhodného čísla. Toto číslo následne rozhodovalo, či sa v metóde `selectAction` zvolí náhodná akcia alebo akcia s najvyššou Q-hodnotou.

6.1.0.6 Trieda SarsaAgent

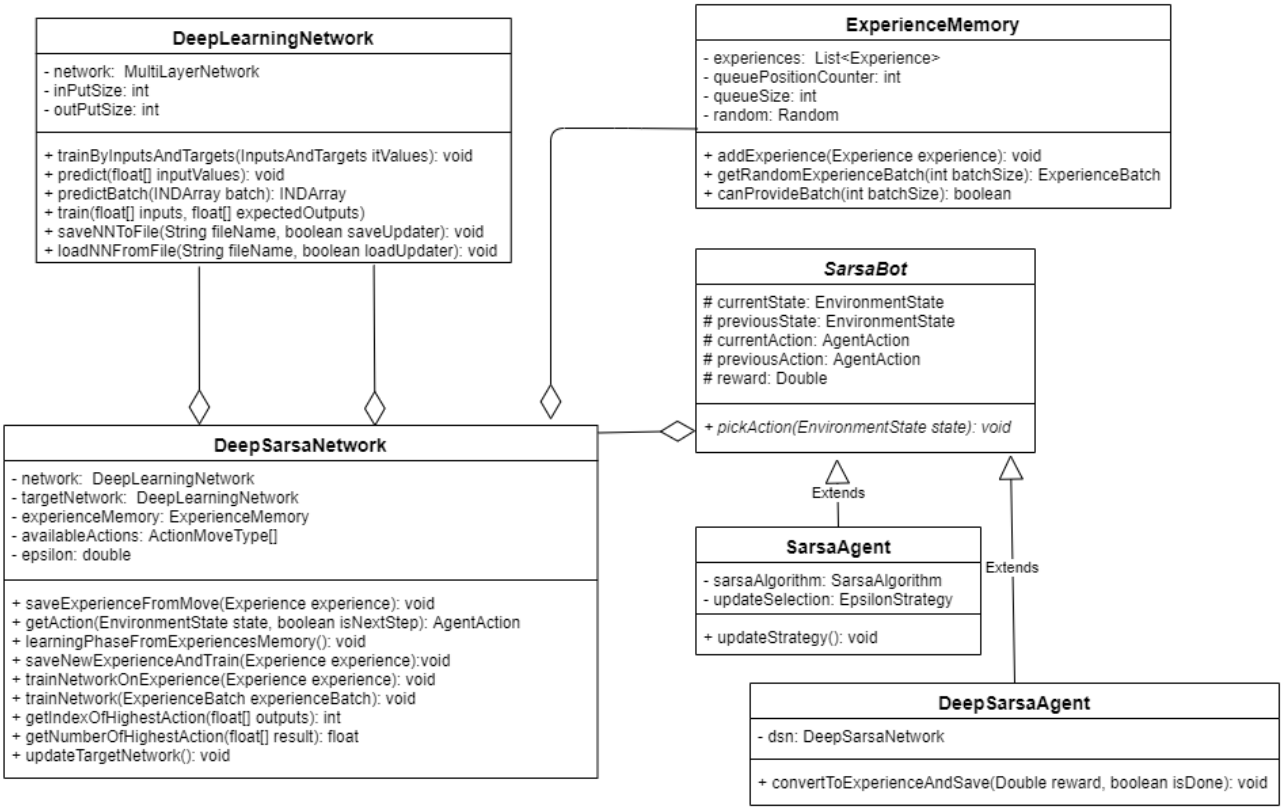
Pre prepojenie učenia a prostredia som implementoval triedu `SarsaAgent`, ktorá slúžila ako agent voliaci akcie v prostredí na základe algoritmu učenia.

Táto trieda využívala triedu `EpsilonStrategy` pre získanie akcie pre daný stav a triedu s názvom `SarsaAlgorithm`, ktorá bola využívaná pre učenie.

Keďže SarsaAgent slúžila iba pre jednoduché *Reinforcement learning* bez umelej neurónovej siete, rozhodol som sa vytvoriť abstraktnú triedu `SarsaBot`, ktorá slúžila ako základ pre agentov jednotlivých učení. V ďalšej implementácii práce som implementoval ďalšiu triedu pre Deep Sarsa learning, ktorá rozširovala túto abstraktnú triedu.

6.2 Implementácia algoritmu SARSA spoločne s neurónovou sieťou

V tejto sekcii je popísaná implementácia Deep SARSA network a jej jednotlivých prvkov za využitia knižnice *Deeplearning4j* (štruktúra implementácie je znázornená na obrázku 6.2).



Obr. 6.2: Triedny diagram pre Deep SARSA network

6.2.0.1 Knižnica Deeplearning4j

Deeplearning4j je *Open Source* knižnica vyvinutá najmä v jazyku Java poskytujúca možnosť vytvorenia Deep Learning sietí. Ponúka vysoký level konfigurovateľnosti parametrov siete a implementácie siete samotnej. Podporuje jazyky ako Java, Scala, Clojure a Kotlin. Táto diplomová práca využívala verziu *1.0.0-BETA7* [12].

6.2.0.2 Trieda DeepLearningNetwork s knižnicou Deeplearning4j

Táto trieda bola jadrom pre samotnú umelú neurónovú sieť a operácie nad ňou, obsahovala tri vlastnosti `inPutSize`, `outPutSize` a `network`. Prvé dve vlastnosti slúžili pre určenie počtu neurónov na vstupnej a výstupnej vrstve. Posledná spomínaná vlastnosť bola typu `MultiLayerNetwork`, ktorý bol štruktúrou siete. Táto vlastnosť sa inicializovala v konštruktore triedy pomocou objektu `NeuralNetConfiguration.Builder`, ktorý slúžil pre 'poskladanie' samotnej siete pomocou jednotlivých vrstiev. Taktiež jednotlivé vrstvy *DenseLayer* využívali triedu `DenseLayer.builder` pre inicializáciu ich jednotlivých vlastností ako:

- `nIn` - počet vstupov do vrstvy;
- `nOut` - počet výstupov z vrstvy;
- `activation` - dátový typ aktivačnej funkcie vrstvy;
- `lossFunction` - dátový typ pre loss function v prípade výstupnej vrstvy;
- `optimizationAlgo` - dátový typ optimizačného algoritmu v prípade vstupnej vrstvy;
- `weightInit` - dátový typ typu incializácie váh v prípade vstupnej vrstvy;
- `updater` - dátový typ aktualizátora v prípade vstupnej vrstvy;

Ďalej som v tejto triede implementoval metódy potrebné pre učenie a predikciu. Metóda `predict` bola používaná pre predikciu hodnôt pre jednotlivé akcie prostredia na základe jej parametru `inputValues`, ktorý reprezentoval stav prostredia. Taktiež bola naimplementovaná metóda `predictBatch`, ktorá mala podobnú funkcionalitu ako predošlá metóda s rozdielom, že predpovedala hodnoty pre dávku z parametru `INDArray batch`.

Pre trénovanie, resp. učenie som implementoval dve metódy. Prvou metódou bola `train`, ktorá slúžila pre naučenie siete na základe vstupu jedného stavu a jeho očakávaných výstupov. Druhou metódou bola `trainByInputsAndTargets`, ktorá učila sieť na základe várky vstupov a ich očakávaných výstupov(parameter `InputsAndTargets itValues`).

Taktiež ako pri triede `QTable` som naimplementoval možnosť ukladať a nahrávať parametre siete. Pre tieto účely vznikli metódy `saveANNToFile`, ktorá uložila parametre do súboru s názvom, ktorý bolo prevzatý z parametru `fileName` a možnosťou taktiež uložiť 'aktualizátor' siete. Metóda `loadANNFromFile` slúžila pre pravý opak a to načítanie parametrov siete zo súboru.

6.2.0.3 Trieda Deep SARSA network

Hlavná trieda celého Deep SARSA learning, slúžila ako reprezentácia oblasti Deep Learning, taktiež obsahovala všetky spomínané vylepšenia. Táto trieda definovala získavanie akcie na základe stavu pomocou ϵ-greedy politiky. Taktiež získavanie a ukladanie skúseností z/do experience memory replay. Veľkú časť implementácie tvorilo najmä učenie a komunikácia s triedou `DepLearningNetwork`.

Vzhľadom k významnosti a komplexnosti jednotlivých metód tejto triedy som sa rozhodol každej venovať vlastný podrobnejší popis.

Metóda `getAction` Táto metóda slúžila pre získanie akcie na základe politiky a stavu. Rozhodol som sa využívať populárnu politiku *ϵ-greedy*, túto politiku som rozšíril o jej modifikáciu `Epsilon decay`, čo znamená, že pri každom zavolaní tejto metódy sa premenná `epsilon` znižovala o hodnotu udanú v konfiguračnom súbore. Hodnota `epsilon` sa znižovala, až kým nedosiahla minimálnu hodnotu určenú v konfiguračnom súbore. Táto hodnota sa vždy porovnávala s náhodne vygenerovaným číslom ak bola väčšia ako toto číslo, zvolila sa náhodná akcia. Ak bola táto hodnota nižšia alebo rovná náhodnému číslu, zvolila sa akcia s najvyššou hodnotou.

Metóda `saveExperienceFromMove` Metóda najprv overovala či `Experience`, ktorá bola príjmaná ako parameter nie je `null`. Nasledovalo vetvenie, ktoré overovalo či bolo použité učenie pomocou eperience memory replay. V prípade učenia s experience memory replay sa pridávala skúsenosť do tejto pamäte metódou `addExperience` a následne sa volala metóda `trainWithERM`. Ak sa nejednalo o učenie s pamäťou, volala sa metóda `trainNetworkExperience`. Mimo vetvenia sa volali ešte dve metódy a to `updateTargetNetwork` a `incrementSteps`.

Metóda `trainNetworkOnExperience` Ak sa Deep SARSA network učila iba pomocou jednotlivých spomienok, využívala sa táto metóda. Prvým krokom bolo vypočítanie Q-hodnoty pomocou metódy `evaluateExperience` pomocou vzorca $y = r$ alebo $y = r + \gamma Q(s', a')$, vetvenie rozhodovalo aký vzorec použiť na základe, či skúsenosť bola terminálna alebo nie. Následne sa táto premenná označovaná triedou `EvaluatedExperience` posielala do metódy `train`, patriacej triede `DeepLearningNetwork`.

Metóda `trainWithERM` Táto metóda sa využívala pri učení s experience replay memory, obsahovala podmienku, ktorá overovala:

- Rovnosť vlastnosti `stepsToLearn` a hodnoty z konfiguračného súboru pre počet krokov pre učenie.
- Zaplnenie skúseností vlastnosti `experienceMemory`.
- Mód trénovania.

Ak sa všetky podmienky splnili zavolala sa metóda `learningPhaseFromExperienceMemory` a metóda `resetSteps`, ktorá vynulovala potrebný počet krokov pre učenie pomocou tejto modifikácie.

Metóda `learningPhaseFromExperiencesMemory` Overovala či vlastnosť `experienceMemory` je schopná dodať dávku náhodných skúseností, ak túto podmienku `experienceMemory` spĺňovala, volala sa metóda `trainNetwork` s parametrom dávky získanej z `experienceMemory`.

Metóda `trainNetwork` Získavala ohodnotenú dávku pomocou algoritmu, ktorý je popísaný na strane 21. Túto dávku následne ako parameter posielala metódou `trainByInputsAndTargets` do triedy `DeepLearningNetwork`.

Metóda `getIndexOfHighestAction` Prvým krokom v tejto metóde bolo overenie jediného parametru `outputs`. Následne sa využíval klasický prístup pre získanie maximálnej hodnoty v poli, čo bolo porovnávanie aktuálnej najväčšej hodnoty a aktuálnej hodnoty elementu v poli `outputs`. Pri porovnávaní sa taktiež prepisoval index elementu s najvyššou hodnotou, ktorý následne táto metóda vracala.

Metóda `updateTargetNetwork` Poslednou významnou metódou bola metóda `updateTargetNetwork`. Jej úlohou bolo overovať pomocou podmienky či sa vlastnosť `stepsToUpdateTargetNetwork` rovnala hodnote z konfiguračného súboru pre počet krokov pre aktualizovanie parametrov siete `Target network`. Taktiež overovala či Deep SARSA network vôbec využíva modifikáciu `Target network`. Ak podmienka bola pravdivá z klasickej `policy` siete sa prekopírovali parametre do `Target network` a vynulovala sa hodnota vlastnosti `stepsToUpdateTargetNetwork`.

6.2.0.4 Target network

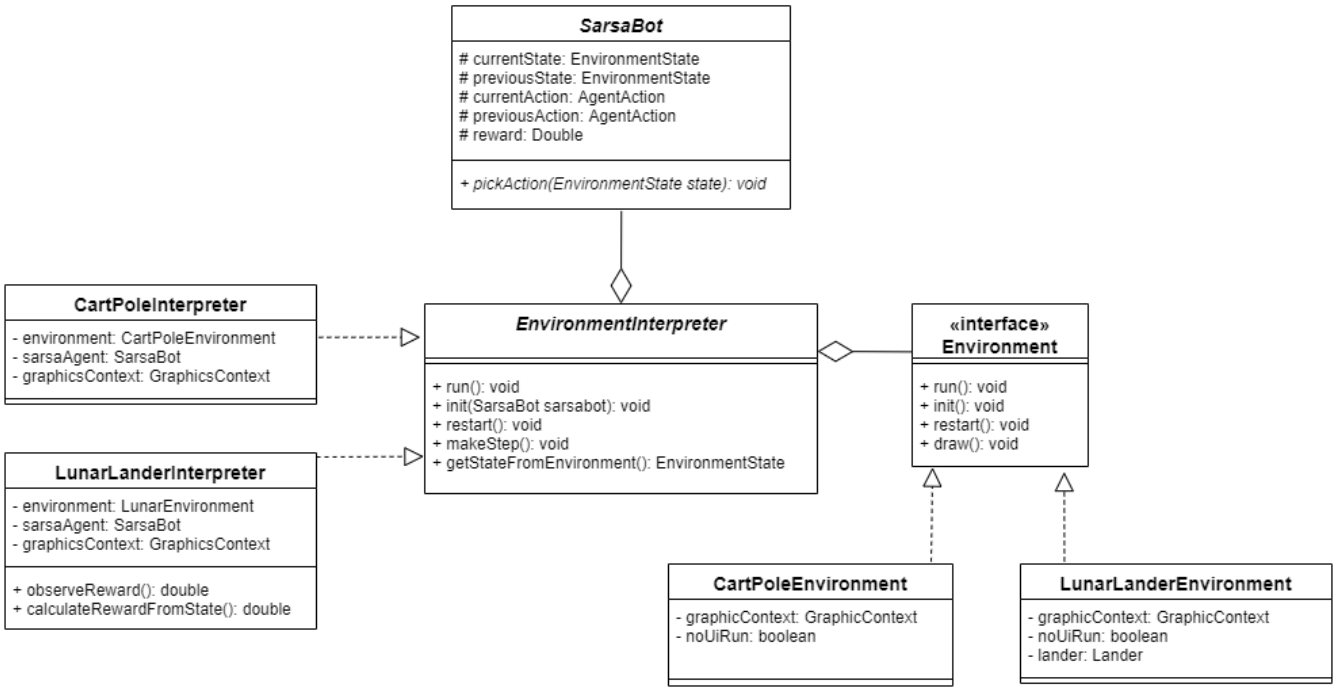
Target network nemala vlastnú špeciálnu triedu pre jej reprezentáciu. Keďže je to sieť úplne istej štruktúry a modelu využil som triedu `DeepLearningNetwork`. Target network bola aktualizovaná pomocou metódy `updateTargetNetwork`.

6.2.0.5 Trieda ExperienceMemory

Pre modifikáciu experience memory replay som naimplementoval triedu `ExperienceMemory`, táto trieda slúžila pre ukladanie skúseností a generovanie náhodných dávok pre učenie Deep SARSA network. Do pamäte sa posúvali skúsenosti zastupované triedou `Experience`. Akonáhle pamäť dosiahla svoju maximálnu kapacitu, nové spomienky nahradzovali staré spomienky. Túto kapacitu určovala vlastnosť `queueSize`.

6.3 Implementácia prostredí a interpretérov

Pre overenie správnej funkčnosti Deep Sarsa učenia som taktiež naiplementoval testovacie prostredia. Jednalo sa o jednoduché prostredia inšpirované prostrediami spoločnosti `OpenAi` [13]. Konkrétne sa jednalo o prostredia *Cart pole balancing* a *Lunar landing*. Tieto prostredia využívali jednoduchú fyziku a boli schopné byť ovládané pomocou agenta algoritmu. O komunikáciu medzi prostredím a agentom sa staral tzv. `Interpreter`, z názvu vyplýva, že intrepretuje akcie agenta do prostredia a následne na základe danej akcie predáva odmenu z prostredia pre agenta.



Obr. 6.3: Triedny diagram interpretéra Prostredia a Agenta

6.3.0.1 Trieda `EnvironmentInterpreter`

Táto abstraktná trieda určovala signatúru pre triedy, ktorého ho implementovali. Jednotlivé triedy teda podľa signatúry využívali predané metódy.

Prostredie a agent učenia sa definovali pomocou metódy `init`. V tejto metóde sa taktiež inicializoval počiatočný stav prostredia, ktorý bol následne predaný agentovi.

Pre spustenie prostredia a zároveň učenia sa využívala metóda `run`. V tejto metóde sa v slučke opakoval postup jednotlivých krokov:

1. Získanie akcie z aktuálneho stavu, pomocou metódy agenta `pickAction`.

2. Vykonanie akcie v prostredí, na základe akcie pomocou metódy `makeStep`.

3. Získanie nového stavu z prostredia pomocou metódy `getStateFromEnvironment`.

4. Získanie akcie z nového stavu, pomocou metódy agenta `pickAction`.

5. Validácia terminálnosti stavu a preskúmanie odmeny.

6. Predanie odmeny agentovi učenia.

   (a) V prípade, že bol stav terminálny, zavolala sa metóda `restart`.

Získavanie aktuálneho stavu pomocou metódy `getStateFromEnvironment` taktiež využívalo na základe prostredia normalizáciu jednotlivých atribútov stavu. Väčšinou sa jednalo o konverziu do rozsahu, lepšie prijateľnému neurónovou sieťou. Teda rozsahu $< 0, 1 >$.

V metóde `restart` sa nastavilo prostredie do počiatočného stavu, taktiež sa priradil počiatočný stav do aktuálneho stavu agenta.

Metóda `makeStep` vykonala v prostredí krok, na základe akcie z agenta. Táto metóda bola zavolaná iba v prípade, že program využíval aj grafický výstup.

6.3.0.2 Rozhranie `Environment`

Pomocou tohto rozhrania bolo jednoduchšie predávať štruktúru metód konkrétnym triedam prostredí, kde tieto metódy boli potrebné pre fungovanie prostredia s interpretérom.

Všetky prostredia využívali triedu `GraphicsContext`, táto trieda umožňovala vykresľovať jednotlivé objekty na plátne grafického výstupu. Vykresľovanie sa vykonávalo v metóde `draw`. Obdobne ako v rozhraní `EnvironmentInterpreter` sa pomocou metódy `init` inicializovali potrebné triedy, taktiež sa vykreslilo celé prostredie s počiatočným stavom.

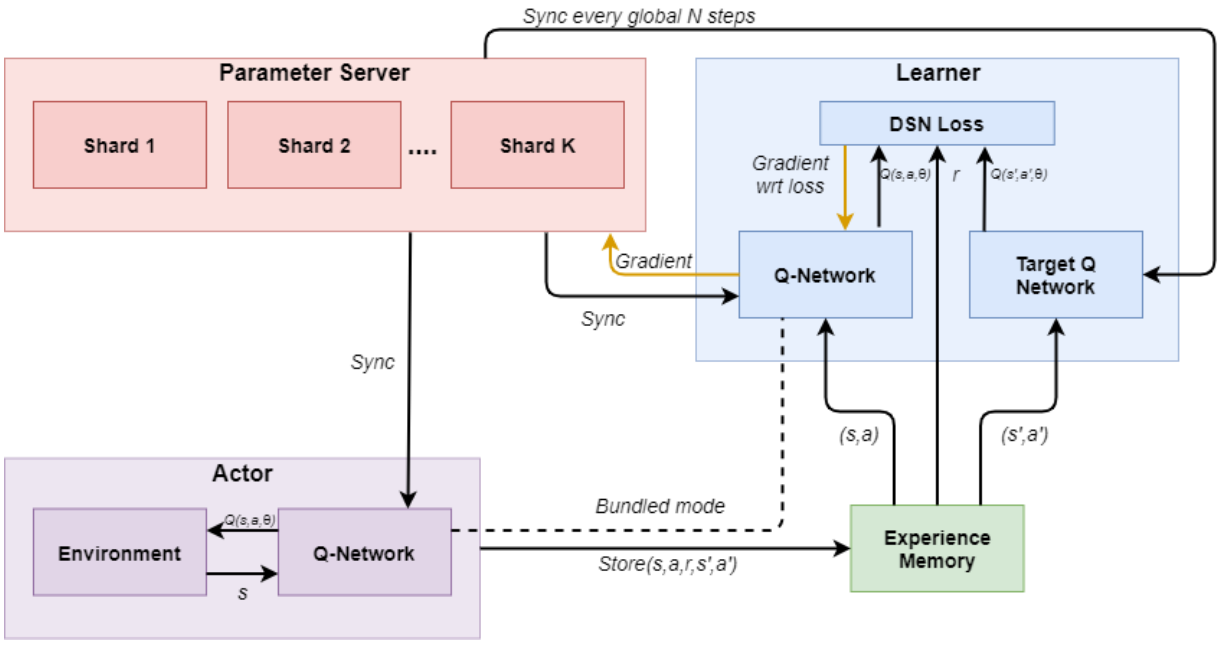
Pomocou vlastnosti `noUiRun` sa overovalo, či sa má dané prostredie vykresľovať, keďže bolo možnosťou taktiež spustiť učenie bez grafického výstupu, čo urýchľovalo učenie alebo umožňovalo paralélne spustenie učenia.

6.4 Gorila architektúra

Vzhľadom k zadaniu diplomovej práce som taktiež implementoval riešenie pre paralélny beh Deep Sarsa učenia na distribuovaných uzloch. Ako jedno z riešení som použil tzv. Gorila(General Reinforcement Learning Architecture) architektúru od spoločnosti Google DeepMind [14]. Táto architektúra využívala Deep Q učenie. Ja som túto architektúru upravil pre Deep Sarsa učenie(obrázok 6.4). Táto architektúra sa skladala zo štyroch hlavných komponent:

- actor;
- global experience memory;

- learner;
- parameter server;



Obr. 6.4: Paralélna architektúra Gorila

Actor

V akomkoľvek Reinfocement learning algoritme agenti volia jednotlivé akcie, na základe aktuálneho stavu. Tento proces sa nazýva *acting*, čiže jednotlivé komponenty vykonávajúce tento proces sa volajú *Actor*. V tejto architektúre sa nachádza N_a odlišných *Actor* procesov, aplikovaných do taktiež N_a príslušných inštancií toho istého prostredia. Každý agent i generuje jeho vlastnú skupinu skúseností s_1^i, a_1^i, r_1^i... s_T^i, a_T^i, r_T^i z prostredia. Tento proces má následne výsledok, že každý *actor* môže navštíviť iné stavy prostredia, čím sa urýchľuje celková explorácia a aj konvergencia.

Každý *Actor* vlastní repliku *Q-network*, ktorá je používaná pre určenie správania, napríklad používaním ϵ-*greedy* politiky. Parametre siete sú periodicky synchronizované s parametrami *Parameter serveru*.

Global experience memory

N-tice skúseností $e_t^i = (s_t^i, a_t^i, r_t^i, s_{t+1}^i, a_{t+1}^i)$ generované pomocou komponent *Actor* sú vkladané do *Experience replay memory D*. V tejto architektúre existujú dva typy týchto modifikácií. Ako prvá sa využíva lokálna *Experience replay memory*, ktorá ukladá každú skúsenosť vygenerovanú pomocou komponenty *actor* lokálne do zariadenia na ktorom je daný *Actor* využívaný. Druhá sa využíva

tzv. *globálna Experience replay memory*, ktorá agreguje skúsenosti do distribuovanej databáze. Pre globálnu *Experience replay memory* platí, že jej veľkosť nezáleží na na počte komponent *Actor*.

Learner

Táto architektúra obsahuje N_l *learner* komponent. Každá komponenta *Learner* vlastní repliku *Q-network* a jeho úlohou je vypočítavať požadované zmeny parametrov *Q-network*. Pre každú aktualizáciu k komponenty *Learner*, je získavaná mini-dávka(mini-batch) z lokálnej/globálnej *Experience replay memory D*, záleží na typu prístupu. *Learner* aplikuje *Deep SARSA network* algoritmus (viz algoritmus 1) na túto mini-dávku skúseností za účelom vygenerovania *sklonu*(gradient) vektoru g_i. Vzostupy g_i sú posielané na *Parameter server* a parametre *Q-network* sú periodicky aktualizované z *Parameter serveru*.

Parameter server

Táto komponenta je používaná ako centrála za účelom udržiavať distribuovanú reprezentáciu *Q-network Q(s, a;* θ^+*)*. Parameter server získáva sklony zo všetkých komponent *Learner* a aplikuje tieto zostupy za účelom modifikovať parametrový vektor θ^+, využítím algoritmu asynchrónneho stochastického sklonového vzostupu.

Gorila architektúra ponúka značnú flexibilitu ako môže byť reinforcement learning agent paralelizovaný. Je možné využívať viacero komponent *actor*, ktoré budú generovať veľký počet skúseností pre globálnu *Experience replay memory* a následne spracovávať tieto skúsenosti pomocou iba jedného sériovej komponenty *Learner*. Naopak je taktiež možné využívať iba jednu komponentu *Actor*, ktorá bude generovať skúsenosti do lokálnej *Experience replay memory* a následne mať viacero komponent *Learner*, ktoré sa paralélne učia na týchto spomienkach. Najzákladnejším módom kombinácie komponent *Actor* a *Learner* je tzv. *Zväzkový mód*(Bundled mode), kde tento zväzok tvorí jedna komponenta *Actor*, *Learner* a lokálna *Experience replay memory*. Jedinú komunikáciu medzi týmito komponentami tvoria parametre siete.

Z tejto architektúry bolo zrejmé, že bolo potrebné použiť určitý typ komunikácie medzi jednotlivými komponentami. Vzhľadom k tomu, že jednotlivé komponenty mohli byť na rozličných distribuovaných uzloch, bolo potrebné zvoliť typ komunikácie, ktorý je schopný viesť komunikáciu cez sieť.

6.4.0.1 Knižnica Aeron

Pre komunikáciu medzi jednotlivými uzlami paralelného učenia bola využitá komunikačná knižnica *Aeron* [15]. Aeron využíva najmä protokol UDP pre unicast ale aj multicast. Bola vyvinutá spoločnosťou Real Logic v jazykoch Java a C++, pre ktoré je tiež určená. Táto knižnica bola využitá

kvôli jej hlavnej vlastnosti a to je výkon, keďže jednotlivé uzly paralelného učenia komunikujú medzi sebou neustále.

Ako komunikačný protokol bol teda využitý protokol UDP, následne som musel zvoliť typ komunikácie. Aeron konkrétne ponúka tri typy komunikácie:

- Unicast - Príjemca komunikácie poslúcha na špecifickom rozhraní (IP adresa) a UDP porte. Príjemca posiela správy o stave a NAK správy (neúspešné prijatie správy) späť odosielateľovi na jeho IP adresu.
- Multicast - Príjemci aj odosielatelia posielajú a príjmajú dáta na rozhranie kontrolného bodu definovaného špecifickou IP multicast adresou(prípadne skupinou) a UDP portom.
- Multi-Destination-Cast - Odosielateľ posiela dáta priamo skupine príjemcov a spravuje si ich ako tzv. Multicast skupinu. Pre komunikáciu využíva Unicast UDP. Táto skupina príjemcov môže byť riadené ako manuálne tak aj dynamicky.

Keďže sa štruktúra paralelného učenia najviac podobala štruktúre Master-Slave(Parameter-server/Global experience memory - Actor/Learner), rozhodol som sa použiť dva typy komunikácii. Pre komunikáciu uzlu Slave s uzlom Master som využil komunikáciu pomocou Unicast. Keďže Master mohol komunikovať s viacerými uzlami typu Slave, využil som typ Multicast, keďže Aeron ponúkal dva typy Multicast komunikácie, musel som zvoliť jeden z nich. Vzhľadom k tomu, že typ Multi-Destination-Cast umožňoval spravovať prihlásených príjemcov, čo znamenalo zachovanie čo najvyššej možnej kontroly, zvolil som tento typ.

Využíval som verziu knižnice 1.32.0 .

6.4.0.2 Implementácia paralelného učenia pomocou Gorila architektúry

Ako bolo v úvode spomenuté využitie architektúry Gorila, pre správne fungovanie tejto architektúry bolo potrebné naimplementovať jednotlivé komponenty tohto prístupu paralelného učenia.

Trieda `ActorRunner` Táto trieda reprezentovala komponentu *Actor*. Rozširovala triedu určenú pre vykonávanie akcií `DeepSarsaAgentActor`. Pomocou vlastnosti `EnvironmentInterpreter`, ktorá bola v rodičovskej triede, mohla vykonávať akcie v danom prostredí. Vzhľadom k odlišnosti komponent a ich využívaní siete, som musel upraviť metódy v rodičovskej triede, ktoré boli využívané pri učení alebo získavaní akcií.

Z diagramu architektúry (viz. 6.4) je zrejmé, že komponenta *Actor* fungovala ako samostatná časť systému. Z tohto dôvodu som využíval pre jej funkcionalitu vlákna, ktoré ponúkali používať bloky kódu paralélne. Trieda `ActorRunner` reprezentovala funkcionalitu pomocou dvoch metód.

Metóda `sendMessagesToParameterServer` slúžila pre zasielanie informácie o počtu vykonaných epizód v danom prostredí na komponentu *Parameter Server*, ktorú reprezentovala vlastnosť `parameterServerClient`, následne boli prijímané parametre zo siete θ^+ komponenty *Parameter*

Server. Druhou metódou bola metóda `sendMessagesToGlobalMemory`, jej úlohou bolo zasielanie nadobudnutých skúseností na komponentu *GlobalExperienceMemory*.

Trieda `LearnerRunner` Táto trieda reprezentovala komponentu *Learner*. Rozširovala triedu reprezentujúcu algoritmus Deep SARSA network, teda triedu `DeepSarsaNetwork` vzhľadom k tomu, že bola využívaná pre získavanie *Gradient* na základe nadobudnutých skúseností.

Využívala metódu `receiveParametersFromParameterServer`, pomocou ktorej aktualizovala parametre siete zo siete θ^+ komponenty *Parameter Server*. V tejto metóde bola volaná metóda `sendGradient`, ktorá posielala `Gradient` na komponentu *Parameter Server*, gradient bol vypočítaný na základe mini-dávky zo získaných skúseností. Druhou metódou pre komunikáciu medzi komponentami bola metóda *receiveExperiencesFromGlobalMemory*, ktorá slúžila pre získavanie skúseností z komponenty *Global Experience Memory*.

Trieda `AeronGlobalExperienceMemory` Pre reprezentáciu komponenty *Global Experience Memory* a komunikáciu medzi ostatnými komponentami bola využívaná trieda *AeronGlobalExperienceMemory*. Jej hlavnou úlohou bolo získavanie skúseností z komponent *Actor* a preposielanie týchto skúseností na komponenty *Learner*.

Obe skupiny komponent boli reprezentované pomocou dátovej štruktúry `HashMap«String, SubscriberStatus»`, kde kľúč reprezentovala adresa na ktorej počúvala daná komponenta a hodnotou bol `enum SubscriberStatus`, ktorý popisoval aktuálny stav danej komponenty vzhľadom ku komunikácii medzi komponentami.

Pomocou metódy `initComponentConnections` sa inicializovali spojenia s komponentami na základe adries získaných pomocou statickej metódy `FileUtil.getIpFromFile` jednotlivých komponent. V metóde `initSendMessageHandler` sa všetkým prihláseným komponentám *Learner* zasielali skúsenosti. Tieto dáta sa vkladali do triedy `LearnerMessage`, ktorá bola následne spracovávaná metódou `handleSendMessage` pre odoslanie na konkrétnu komponentu. Správy, ktoré obsahovali skúsenosti z komponent *Actor* boli spracovávané v metóde `initReceiveMessageHandler`, ktorá inicializovala triedu `FragmentHandler`, ktorá slúžila pre spomínané spracovanie príchodzích správ.

Trieda `AeronParameterServer` Táto trieda bola naimplementovaná podobne ako trieda pre komponentu *Experience memory replay* `AeronGlobalExperienceMemory`, keďže fungovali v architektúre podobným spôsobom. Odlišovala sa od triedy `AeronGlobalExperienceMemory` komunikáciou s komponentou *Actor*, kde obe komponenty *Actor* aj *Learner* prijímali parametre siete Q^+. Pre toto zasielanie parametrov sa využívali metódy `handleLearnerMessaging` a `handleActorMessaging`.

Taktiež táto trieda príjmala správy z oboch spomínaných komponent, z komponenty *Actor* príjmala počet dosiahnutých epizód, tento počet slúžil pre porovnávanie s ukončovacím parametrom `trainingEpochs`, ktorý bol získavaný z konfiguračného súboru. Nasledujúcim typom prijímanej správy, bola správa z komponenty *Learner*, ktorá obsahovala `Gradient` siete danej komponenty.

Trieda `ParameterServerRunner` V tejto triede, ktorá bola vlastnosťou triedy `AeronParameterServer` sa vykonávali konkrétne operácie s prijatými gradient. Taktiež sa získavali parametre zo siete Q^+ komponenty *Parameter server*, ktoré následne boli zasielané pre synchronizáciu na komponenty *Actor* a *Runner*.

Pomocou metódy `updateNetworkFromGradient` sa aktualizovala sieť Q^+ na základe gradient. Aktualizácia mohla byť vykonaná pomocou troch možných prístupov:

- *Average* - komponenta *ParameterServer* čakala, kým prijala gradient zo všetkých komponent typu *Learner*, následne jednotlivé hodnoty vektoru získaného z objetku triedy `Gradient` boli spriemerované a aplikované do siete.
- *Sum* - obdobne ako v prístupe *Average*, komponenta čakala na gradient, rozdiel tvorila matematická operácia, ktorou bola suma jednotlivých hodnôt vektoru gradient.
- *Plain* - v tomto prístupe komponenta nečakala na zvyšné gradient, akonáhle prijala nejaký gradient, aplikovala ho na sieť.

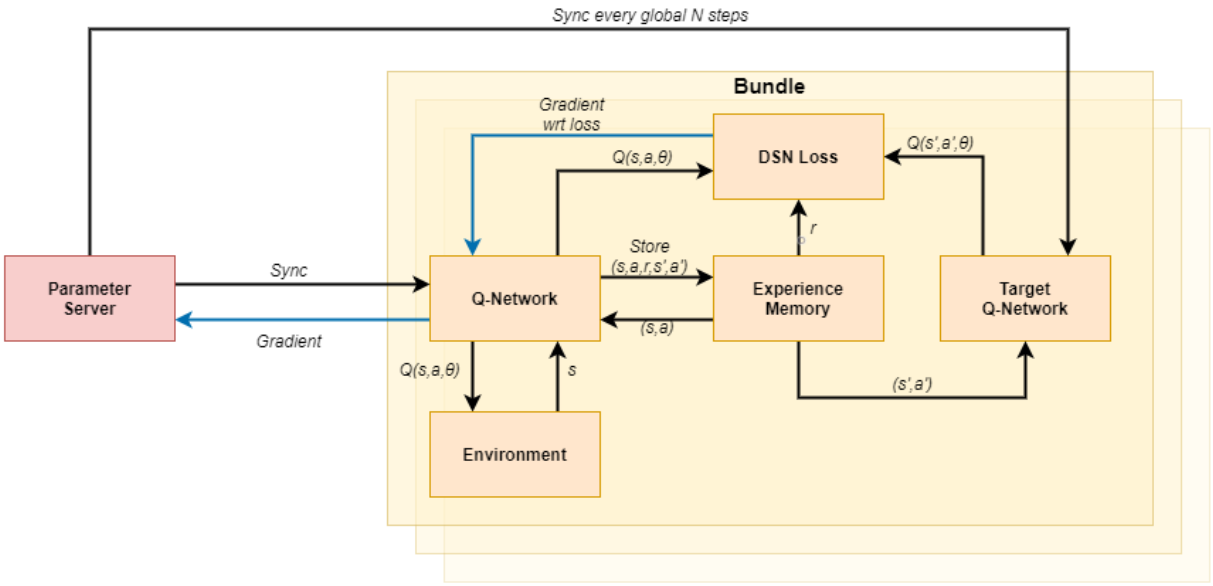
Druhou podstatnou metódou bola metóda `getParams`, ktorá získavala parametre siete Q^+.

Trieda `Message` Táto trieda slúžila pre preposielanie dát medzi jednotlivými komponentami, keďže správy obsahovali rôzne dáta, táto trieda bola abstraktná a obsahovala iba vlastnosti, ktoré boli potrebné v každom type správy.

Rozširovali ju triedy: `LearnerMessage` a `ActorMessage`, ktoré obsahovali dodatočné vlastnosti potrebné pre komunikáciu daných komponent. Pre označenie odosielateľa a prijímateľa správy som použil `MessageType`.

6.5 Bundle architektúra

Vzhľadom k pomerne vysokej labilite paralelného učenia pomocou architektúry Gorila som sa rozhodol využiť jednoduchší prístup. Architektúra tohto učenia bola založená na dvoch komponentách: *Parameter Server* a *Bundle* (obrázok 6.5). Táto architektúra bola inšpirovaná *Bundled* módom používaným v architektúre *Gorila*.



Obr. 6.5: Architektúra Bundle

Parameter Server

Táto komponenta fungovala takmer takisto ako v architektúre Gorila. Odstránila sa v nej komunikácia s komponentou *Actor*, ktorá je zahrnutá v komponente *Bundle*. Jej hlavnou úlohou bolo teda spracovanie gradient z jednotlivých komponent *Bundle* a posielanie parametrov jej siete θ^+ jednotlivým komponentám.

Bundle

V porovnaní s architektúrou Gorila, kde mali viaceré komponenty odlišné špecifické úlohy sa v architektúre *Bundle* využíval tzv. zväzok týchto komponent v jednej komponente. Tým som dosiahol zníženie potrebnej komunikácie medzi jednotlivými komponentami, ktorá fungovala asynchrónne a tým pádom vytvárala pri väčšom počte komponent zmätok pri spracovávaní správ.

Komponenta *Bundle* teda zahŕňala komponenty *Actor* a *Learner*. V architektúre neexistovala globálna Experience memory, keďže sa spomienky priamo posielali z prostredia do lokálnej Experience memory, pomocou ktorej následne komponenta vytvorila gradient pre *Parameter Server*.

6.5.0.1 Implementácia paralelného učenia pomocou *Bundle* architektúry

Implementácia tejto architektúra bola jednoduchšia vzhľadom k počtu komponent a ich samotnej komplexnosti. Existovali dve triedy tejto architektúry `LearningBundle` a `AeronBundleParameterServer`.

Trieda `LearningBundle` Táto trieda reprezentovala komponentu *Bundle* vzhľadom k tomu, že musela získavať skúsenosti z prostredia, rozširovala triedu `DeepSarsaAgent`, ktorá fungovala ako prostredník medzi učením *Deep Sarsa* a samotným prostredím. Kvôli potrebe samostatného behu tejto komponety, trieda implementovala rozhranie `Runnable`, ktoré umožňovalo pomocou implementácie metódy `run` beh na vlákne.

V metóde `run` sa vykonávala celá logika tejto triedy. Pomocou metódy `initParametersFromServer` sa inicializovali parametre sietí Q a Q^- zo siete Q^+ komponenty *Parmeter Server*. Vďaka metóde `makeStep` sa následne sa vykonal krok v danom prostredí a uložila sa získaná skúsenosť do experience memory. Vypočítaný gradient siete sa následne zasielal na parameter server pomocou metódy `sendGradientToServer`, tento gradient bol vkladaný do triedy `BundleMessage`, ktorá bola používaná pre prenos dát medzi komponentami. Posledným krokom bolo prijímanie parametrov z komponenty *Parameter Server*. Tieto parametre boli získavané v metóde `receiveParametersFromServer`, konkrétne sa volala metóda `receiveMessage` vlastnosti `parameterServer`, ktorá reprezentovala komunikáciu s komponentou *Parameter Server*.

Trieda `AeronBundleParameterServer` Touto triedou bola reprezentovaná komponenta *Parameter Server*. Implementácia triedy bola podobná ako v prípade architektúry Gorila. Hlavnou úlohou tejto triedy bolo zasielanie parametrov svojej siete Q^+ na jednotlivé komponenty *Bundle* a samozrejme taktiež prijímanie spracovanie získaných gradient zo spomínaných komponent. Táto trieda vlastnila vlastnosť `ParameterServer` ktorá priamo spracovala gradient a získavala parametere siete Q^+.

Jediným hlavným rozdielom medzi triedou `AeronBundleParameterServer` a triedou architektúry *Gorila* `AeronParameterServer` bola komunikácia vzhľadom ku komponentám. Na komponentu *Bundle* sa zasielali parametre a taktiež sa z nej prijímali gradient, kde v triede `AeronParameterServer` boli tieto akcie rozdelené medzi komponenty *Actor* a *Learner*.

6.6 Implementácia užívateľského prostredia pre modul SARSA v aplikácii NeuronNetModeler

6.6.0.1 NeuronNetModeler

Celú svoju diplomovú prácu som implementoval ako samostatný modul pre aplikáciu *NeuronNetModeler*. Táto aplikácia mala dlhodobý cieľ, čo bolo vytvorenie simulátoru neurónových sietí s podporou vizualizácie vnútornej štruktúry, priebehu učenia a jeho výsledkov. Spomínaný simulátor bol určený predovšetkým pre podporu výuky predmetu *Neuronové sítě*. Na tomto projekte sa podieľalo veľa študentov a taktiež tento projekt obsahoval širokú škálu typov neurónových sietí a ich implementácií, ktoré reprezentovali diplomové práce a semestrálne práce študentov.

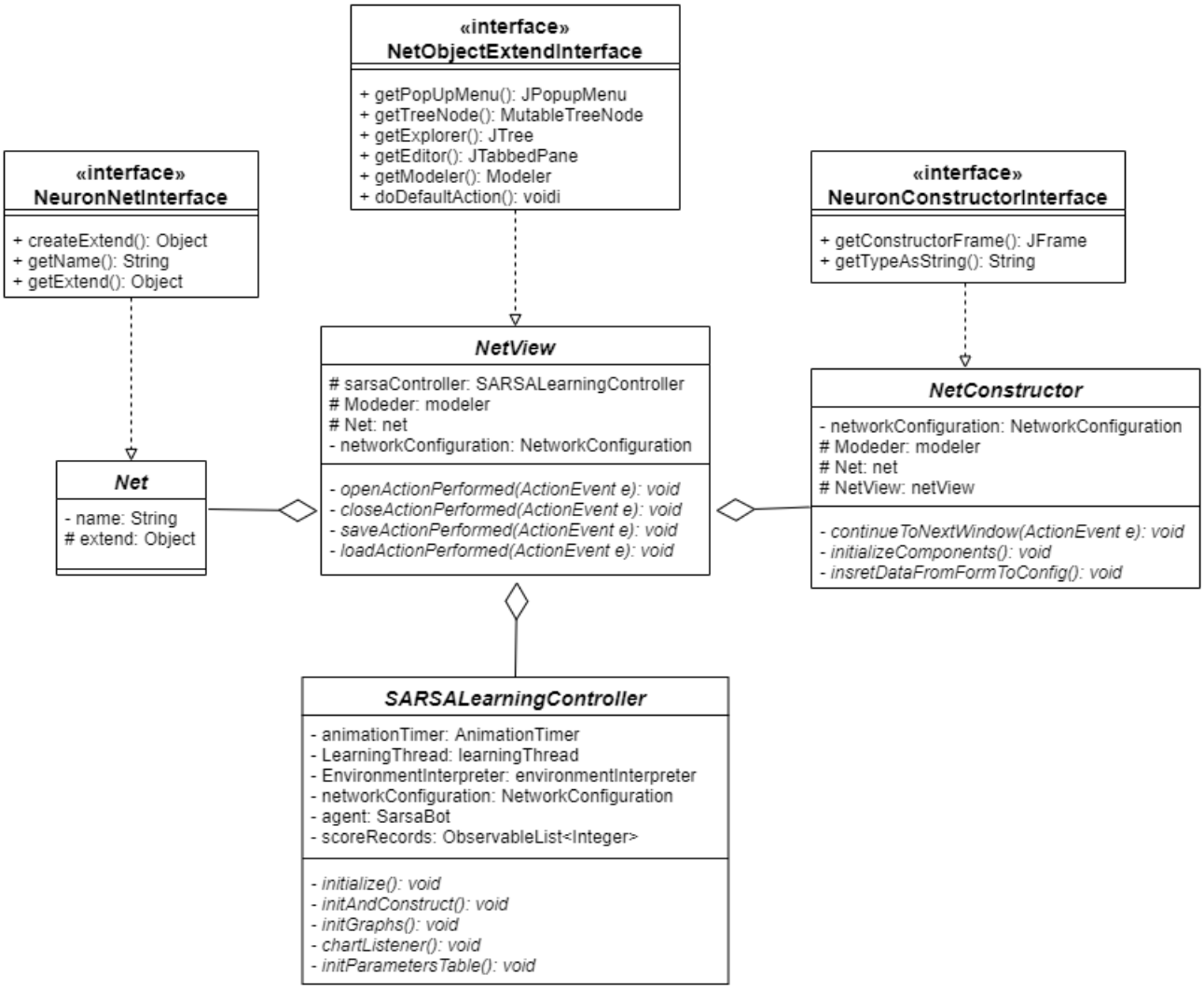
6.6.0.2 Základná štruktúra

Pre správne fungovanie a zaimplementovanie môjho modulu bolo potrebné implementovať rozhrania ponúkané projektom *NeuronNetModeler*(štruktúra implementácie je na obrázku 6.6). Jednalo sa o rozhrania:

- `NetConstructorInterface` - Rozhranie, ktoré popisovalo metódy potrebné pre vytvorenie siete a jej doplnkov.
- `NetObjectExtendInterface` - Rozhranie, ktoré popisovalo v ktorom sa nachádalo samotné učenie siete.
- `NeuronNetInterface` - Rozhranie, ktoré popisovalo metódy potrebné pre prácu so samotnou sieťou.

Trieda `Net` Úlohou tejto triedy bolo priamo pracovať s konkrétnou neurónovou sieťou. V prípade mojej implementácie nebolo veľa metód kompatibilných s implementáciou pomocou knižnice *DeepLearning4J*. Takže som implementoval iba najzákladnejšiu metódu `createExtend`, ktorá slúžila pre skonštruovanie `Class` objektu pre triedu `NetView`.

Trieda `NetView` Táto trieda reprezentovala okno v ktorom sa priamo zobrazovalo učenie siete, prípadne algoritmu. Toto okno bolo skonštruované pomocou sady nástrojov pre grafické rozhranie `Swing`. Konkrétne trieda `NetView` rozširovala triedu `JPanel`. Táto trieda bola využívaná teda pre priamu prácu s modulom, ktorý reprezentovala trieda `SARSALearningController`. Keď sa trieda vytvorila pomocou triedy `NetConstructor`, inicializovalo sa menu pre prácu s modulom. Toto menu ponúkalo základné akcie nad modulom ako: `Open, Save, Load, Close, Train`. Akonáhle sa zaregistroval klik myšou na položku `Open`, zavolala sa pomocou `ActionListener` metóda `openActionPerformed`, kde sa inicializovala trieda `SARSALearningController`.



Obr. 6.6: Triedny diagram implementácie modulu

Trieda `NetConstructor` Pomocou triedy `NetConstructor`, ktorá rozširovala taktiež triedu `JPanel` sa inicializovali počiatočné parametre siete, teda učenia SARSA. Tieto parametre boli načítané z triedy `NetworkConfiguration`. Táto trieda držala všetky potrebné parametre pre učenie, bolo ju možné načítať zo súboru typu `.yml`. Taktiež obsahovala bezparametrický konštruktor, ktorý poskytoval základne parametre. Tieto parametre boli zobrazované pomocou triedy `JTextField`.

Pre prechod do okna učenia sa používala metóda `continueToLearning`, ktorá vložila všetky zmenené hodnoty do objektu triedy `NetworkConfiguration`. Následne vytvorila novú inštanciu `Net` a `NetView` a zavolala metódu `insertNet` objektu `modeler`.

Trieda `SARSALearningController` Trieda `SARSALearningController` slúžila pre priame prepojenie aplikácie `NeuronNetModeler` a môjho modulu `reinforcementLearningSarsa`. Narozdiel od predošlých tried využívala sadu nástrojov z platformy *JavaFx*. Táto trieda taktiež spravovala logiku nad grafickým rozhraním, ktoré bolo definované v súbore `sarsa-controller.fxml`.

Pomocou metódy `initAndConstruct` sa inicializovali potrebné vlastnosti tejto triedy. Na základe názvu prostredia, ktoré bolo získavané z objektu triedy `NetworkConfiguration` sa inicializoval konkrétny intepretér. Obdobne sa taktiež inicializoval *agent*, keďže každý *agent* dokázal spolupracovať s daným prostredím iným spôsobom.

Následne sa inicializovali všetky potrebné grafické elementy pomocou metód `initParameterstable` a `initGraphs`. Akonáhle sa zaregistrovala akcia z tlačidla `startButton`, pomocou novo vytvorenej inštancie triedy `AnimationTimer` a jej metódy `handle` sa vytvorila slučka. V tejto slučke sa volala metóda `run` daného interpretéra, čím sa začalo konkrétne učenie.

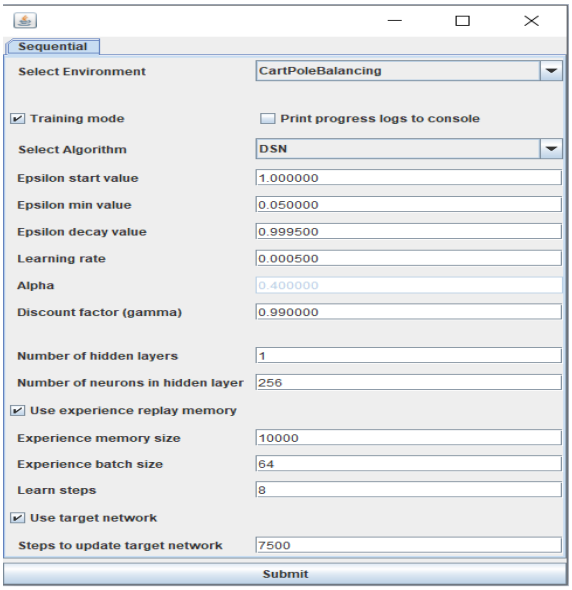
V prípade vyvolania metódy `accelerateButtonPressed` na základe užívateľskej akcie sa spomínaná inštancia triedy `AnimationTimer` zastavila. Metóda `run` bola následne využívaná pomocou novej inštancie triedy `LearningThread`, ktorá rozširovala triedu `Thread`, čiže učenie spustila na vlákne až kým sa dané vláknom nezastavilo.

Kapitola 7

Užívateľská príručka

V rámci spracovania diplomovej práce som taktiež implementoval jednoduché užívateľské rozhranie, ktoré rozširovalo program *NeuronNetModeler* o nový modul zvaný *DeepSARSALearning*.

Po zvolení spomínaného modulu sa zobrazilo okno pre nakonfigurovanie jednotlivých parametrov algoritmu(obrázok 7.1). Ako prvé bolo potrebné zvoliť trénovacie prostredie. Následne si užívateľ mohol vybrať konkrétny algoritmus. Na základe tejto voľby sa sprístupnili, respektíve zakázali polia konkrétnych parametrov, ktoré pre daný algoritmus neboli vyžadované.

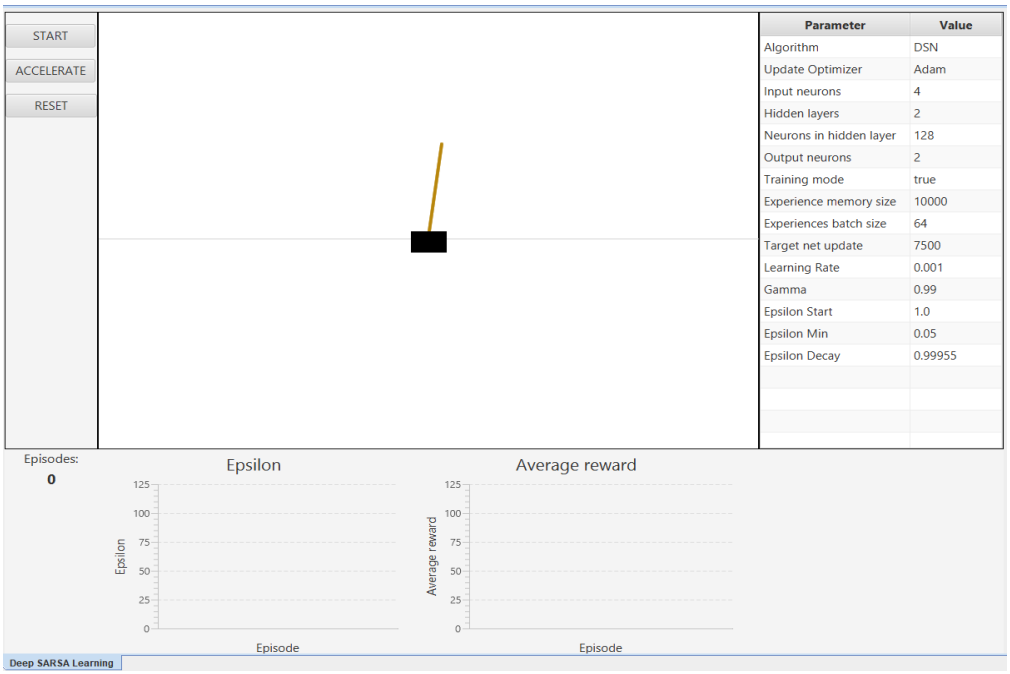


Obr. 7.1: Ukážka nástroju pre konfiguráciu učenia algoritmu

Zakliknutím tlačidla *Submit* sa potvrdila zvolená konfigurácia algoritmu a prostredia. Po tejto akcii sa zobrazilo hlavné okno, ktoré obsahovalo v ľavej hornej časti vytvorený algoritmus. Pravým kliknutím myši na tento objekt sa zobrazilo kontextové menu, ktoré obsahovalo jednotlivé možnosti pre prácu s algoritmom.

Zvolením tlačidla *Open* sa otvorila hlavná obrazovka nástroju pre učenie algoritmu. V tejto obrazovke(obrázok 7.2) bolo možné pozorovať priebeh učenia algoritmu či už priamo jeho riešenie v grafickom okne, alebo pomocou prístupných grafov, ktoré mapovali jeho dosiahnutú odmenu a aktuálnu hodnotu explorácie(hyperparameter epsilon). V pravej časti tohto okna boli zobrazené všetky parametre zvolené na začiatku. Program umožňoval správu učenia pomocou troch tlačidiel:

- *START/STOP* - Štart alebo pozastavenie učenia.
- *ACCELERATE/DECELERATE* - Pomocou tohto tlačidlo bolo možné urýchliť učenie, akonáhle sa zakliklo toto tlačidlo grafické okno učenia sa zamrazilo a učenie bolo spustené na novom vlákne, opätovným stlačením sa učenie vrátilo do základnej rýchlosti a zobrazovalo sa v grafickom okne.
- *RESET* - Obnovenie učenia do pôvodných nastavení.



Obr. 7.2: Ukážka nástroju pre učenie algoritmu

Ďalšími možnosťami v kontextovom menu bolo načítanie parametrov učenia pre dané prostredie pomocou tlačidla *Load*, ktorým zakliknutím bol užívateľ vyžiadaný pre zvolenie súboru pre načítanie parametrov.

Opakom tejto akcie bolo tlačidlo *Save*. Pomocou tohto tlačidla bol užívateľ schopný uložiť parametre učenia do súboru.

Pomocou tlačidla *Train* bolo možné taktiež spustiť učenie.

Tlačidlá *Open* a *Close* slúžili pre otvorenie, respektíve zatvorenie obrazovky hlavného nástroja pre učenie.

Kapitola 8

Experimenty

Táto kapitola popisuje rôzne typy experimentov nad rôznymi prostrediami na základe využitia spomínaných algoritmov a neurónových sietí spoločne s ich modifikáciami. Experimenty sa rozdeľujú do skupiny sekvenčného behu v lokálnom prostredí a do skupiny, ktorá popisuje paralélny beh na uzloch superpočítača.

8.1 Sekvenčný beh

Cieľom experimentov v tejto skupine bolo otestovanie funkčnosti implementácie učenia *SARSA* a *Deep SARSA*. Následne pomocou úprav hodnôt jednotlivých hyperparametrov a využitia jednotlivých modifikácii učenia som porovnával rýchlosť a efektivitu jednotlivých konfigurácii, na základe týchto výsledkov som bol schopný následne porovnať rozdiely medzi paralelným učením a sekvenčným. Cieľom bolo teda zistiť možné zrýchlenie v paralélnom učení.

Experimenty v sekvenčnom typu behu boli vykonávané na lokálnom prostredí. Toto prostredie bolo reprezentované mojim osobným počítačom značky ASUS.

Názov	ASUS FX503VD-E4082T
OS	Windows Home 10 v20H2
CPU	Intel(R) Core i5-7300HQ 2.50GHz
CPU jadrá	4
RAM	16 GB
Disk	Samsung SSD 970 EVO Plus 250GB

Tabuľka 8.1: Konfigurácia zariadenia použitého pre sekvenčné experimenty.

Experimenty sa ďalej vykonávali v dvoch testovacích prostrediach, *Cart Pole balancing* a *Lunar Landing*, tieto prostredia predstavovali jednoduché fyzické výzvy pre algoritmy, ktoré sa ich snažili riešiť. Tieto prostredia som implementoval v jazyku *Java* za využitia grafickej sady nástro-

jov *JavaFX*. Ako príklad pre správnu implementáciu som využil testovacie prostredia ponúkané organizáciou *OpenAI*.

8.1.1 Cart Pole balancing

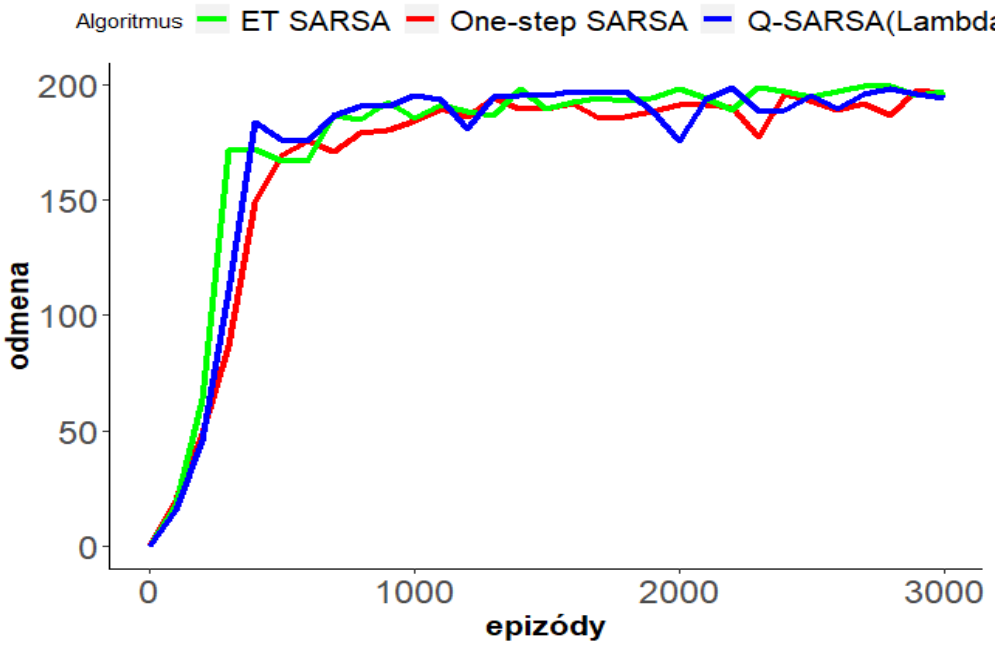
Prostredie *Cart Pole balancing* bolo využívané a testované najviac vzhľadom k jeho pomerne jednoduchému procesu riešenia. Experimenty boli najprv vykonávané nad SARSA učením a jeho modifikáciami *Eligibility traces* a $Q - SARSA(\lambda)$. Následne som testoval a vykonával experimenty nad Deep Sarsa učením a jeho modifikáciami ako *Experience replay memory* a *Target Network*.

8.1.1.1 SARSA učenie

Prvá sada experimentov bola vykonaná nad učením *SARSA* podľa nasledovných parametrov:

Parametry SARSA učenia	
Gamma	0.99
Alpha	0.4
Epsilon start	1.0
Epsilon min	0.05
Epsilon decay	0.99955

Tabuľka 8.2: Parametry SARSA učenia pre Cart Pole Balancing



Obr. 8.1: Graf SARSA učenia v prostredí Cart Pole balancing

Vzhľadom k celkom komplexnému počtu stavov tohto prostredia som musel pre algoritmy SARSA učenia normalizovať vstupy, čím som dosiahol menšieho počtu týchto stavov a rýchlejšieho a efektívnejšieho riešenia prostredia. Táto normalizácia bola vykonávaná pomocou rozdelenia stavov do tzv."boxov".

Z grafu 8.1 je viditeľné, že všetky typy algoritmov boli schopné vyriešiť toto prostredie. Učenie všetkých algoritmov bolo pomerne rýchle, algoritmy dosiahli odmeny vyššej ako 190 po vyše 1000 epizódach. Ku konvergencii ku správnemu riešenie došlo po vyše 200 epizódach u všetkých algoritmov.

Z grafu 8.1 je taktiež možné vidieť, že najstabilnejším algoritmom bol *Eligibility Traces SARSA*, ktorý síce konvergoval o niečo pomalšie, ale udržiaval si stály stabilný rast v učení. V tomto učení mu pomohlo využívanie tzv.*Eligibility Traces*, ktoré využívajú pre učenie predošlé kroky agenta.

8.1.1.2 Deep SARSA učenie

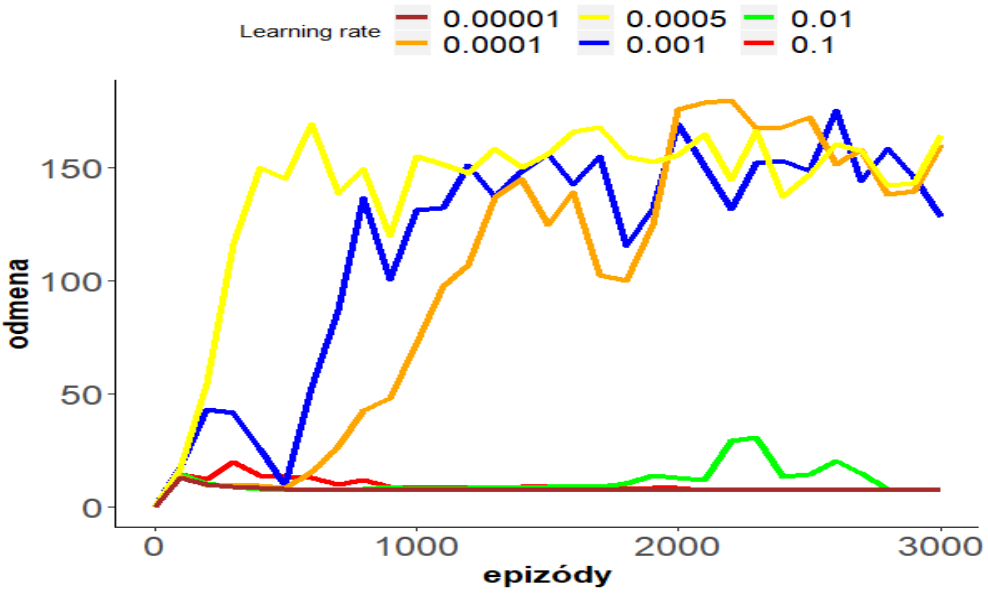
Ďalšie experimenty boli vykonané nad Deep SARSA učením a následne aj nad kombináciami jeho modifikácii. Prvým experimentom bol teda samostatný algoritmus Deep SARSA bez akýchkoľvek modifikácii. Behom týchto experimentov som postupne menil rôzne hyperparametre pre nájdenie najvhodnejšej konfigurácie tohto algoritmu.

Ako prvý parameter som upravoval hyperparameter *Learning rate*. Začal som hodnotou *0.1* a túto hodnotu som pre každý experiment následne zmenšoval na stotinu predošlej hodnoty. Pre tento experiment bola využitá nasledovná konfigurácia siete.

Update optimizer	Adam
Gamma	0.99
Počet skrytých vrstiev	2
Počet neurónov v skrytej vrstve	128
Epsilon start	1.0
Epsilon min	0.05
Epsilon decay	0.99955

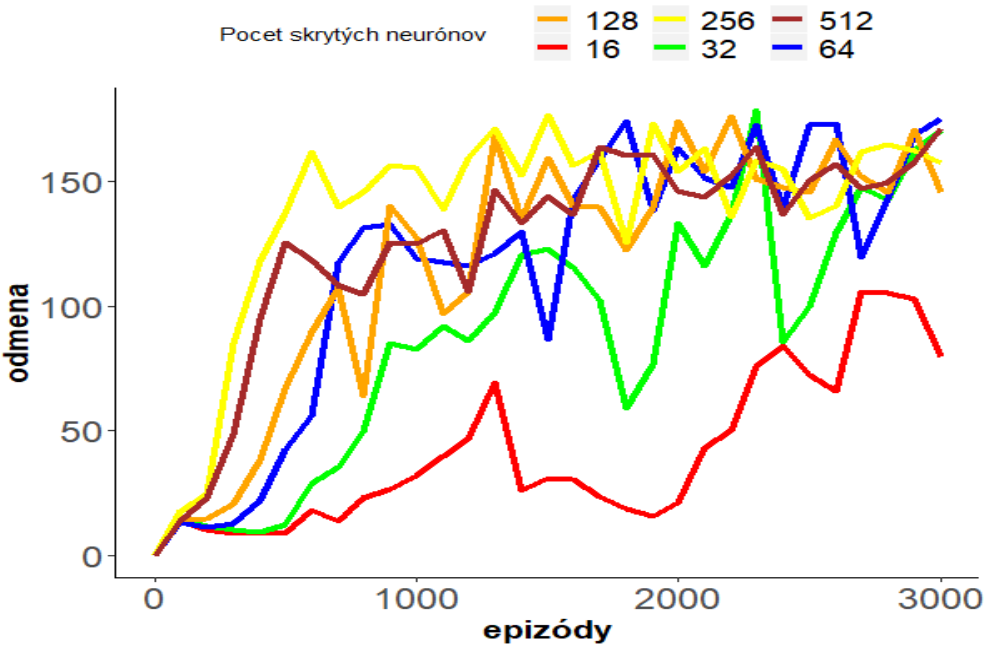
Tabuľka 8.3: Parametry Deep SARSA učenia pre Cart Pole Balancing

V nasledovnom grafe 8.2 je vidieť, že hyperparemeter *Learning rate* má vysoký vplyv na učenie algoritmu. Veľmi vysoké a taktiež veľmi nízke hodnoty tohto hyperparemetru viedli algoritmus k zlému učeniu, respektíve algoritmus nebol schopný riešiť prostredie. Najefektívnejšou hodnotou bola hodnota *0.0005*, vďaka ktorej algoritmus rýchlo konvergoval a dosiahol najväčšej stabilnej odmeny.



Obr. 8.2: Graf SARSA učenia v prostredí Cart Pole Balancing pri zmene Learning Rate

Keďže som poznal vďaka predošlému experimentu najlepšiu hodnotu hyperparametru *Learning rate*, mohol som sa presunúť na otestovanie najvhodnejšieho počtu neurónov v skrytej vrstve. Experimenty začínali na hodnote *16*, kde som následne túto hodnotu upravoval pre každý experiment ďalšou mocninou čísla *2*.



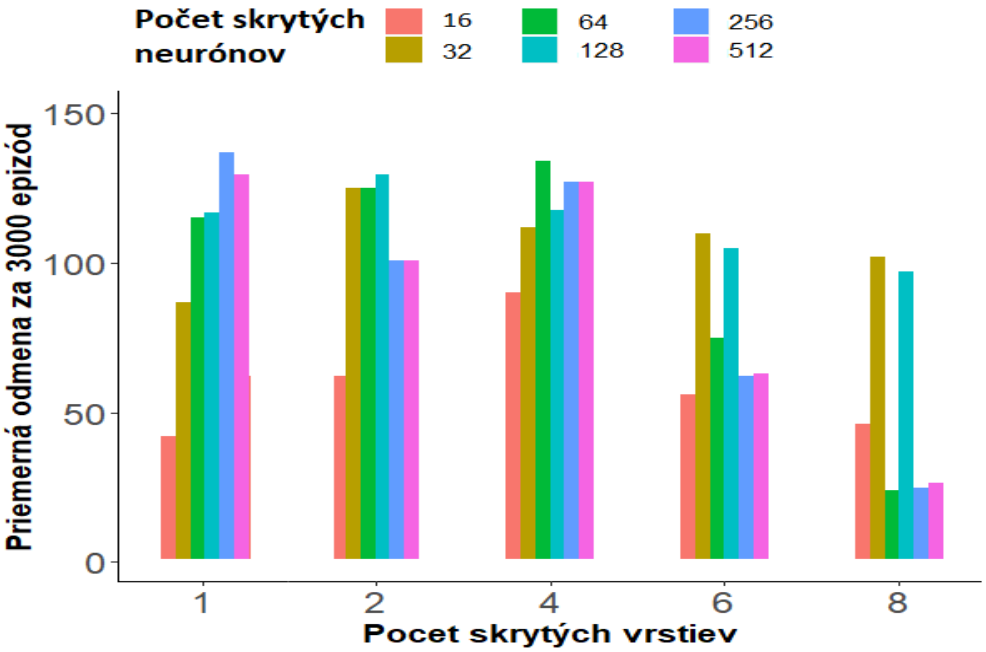
Obr. 8.3: Graf Deep SARSA učenia v prostredí Cart Pole balancing pri zmene počtu neurónov

Z grafu 8.3 je teda badateľné, že vyšší počet neurónov v skrytej vrstve zvyšoval výšku získanej odmeny. Kým nízky počet neurónov smeroval učenie neurónu k nižšej odmene alebo v lepšom prípade pomalšej konvergencii, vysoký počet zvyšoval konvergenciu a priemernú odmenu. Najvhodnejším počtom neurónov v skrytej vrstve sa ukázali byť dve hodnoty a to *256* a *512*.

Jedinou nevýhodou zvyšovania počtu neurónov bol čas učenia. Čím sa počet neurónov v skrytej vrstve zvyšoval, tým sa taktiež predlžovala doba učenia a samozrejme taktiež doba pre vykonanie *3000* epizód.

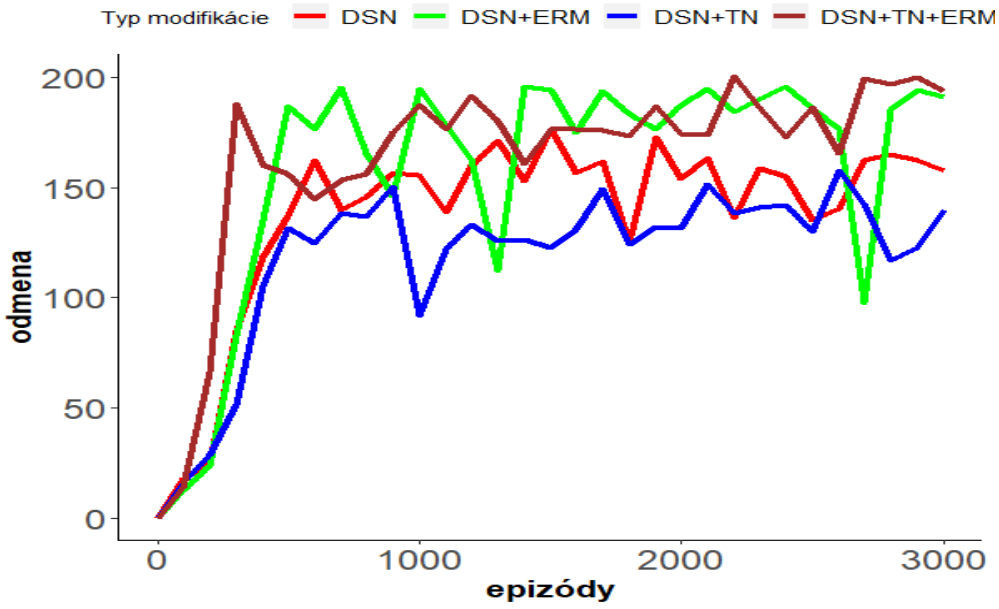
Ako bolo popísané v kapitole o Deep Learning (viz. strana 20), siete tohto prístupu často obsahujú viac než jednu skrytú vrstvu, práve kvôli čomu sa volajú *Deep*. Úlohou ďalšieho experimentu bolo teda zistiť aký počet skrytých vrstiev je najvhodnejší pre riešenie prostredia *Cart Pole balancing*.

Z nasledujúceho grafu 8.4 je teda viditeľné, že pre všetky možnosti počtu neurónov v skrytej vrstve je vhodné použiť jednu až štyri vrstvy. Pre ešte vyšší počet skrytých vrstiev začína byť sieť neefektívna v učení. Najvhodnejšou kombináciou sa ukázala byť kombinácia vrstvy a neurónov. Túto kombináciu som využíval pre ďalšie experimenty.



Obr. 8.4: Graf efektivity skrytých vrstiev

Vďaka všetkým predošlým experimentom som získal najvhodnejšiu konfiguráciu pre vykonávanie ďalších experimentov s modifikáciami Deep SARSA. Experimenty som začal samotným algoritmom a následne som využíval kombinácie modifikácií *Target Network* a *Experience memory replay*.



Obr. 8.5: Graf efektivity modifikácii Deep SARSA network

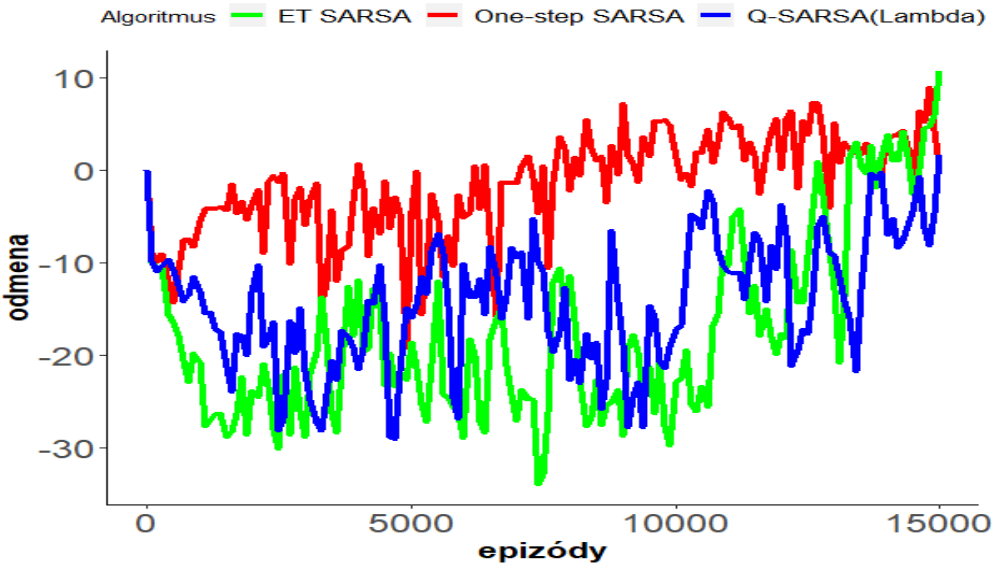
Z grafu 8.5 je vidieť, že samotný algoritmus nie je schopný dosiahnuť požadovanej odmeny do počtu *3000* epizód. Využitie algoritmu spoločne s modifikáciou *Target Network* nebolo vhodné, keďže sa dosiahnutá priemerná odmena znížila oproti využitiu základného algoritmu. Naopak významné zlepšenie učenia bolo dosiahnuté vďaka modifikácii *Experience Replay memory*. Vďaka kombinácii všetkých modifikácii bolo dosiahnuté efektívne učenie a zároveň bola zvýšená stabilita.

8.1.2 Lunar Landing

Ďalšími experimentami boli experimenty vykonané nad komplexnejším prostredím *Lunar landing*.

8.1.2.1 SARSA učenie

Vzhľadom ku veľkosti počtu možných stavov v tomto prostredí bolo potrebné vykonávať normalizáciu stavov. Táto normalizácia bola nutná pre základný algoritmus SARSA, ktorý by inak nebol schopný riešiť dané prostredie. Oproti prostrediu *Cart Pole balancing* trvalo učenie podstatne dlhšie, respektíve, trvalo *15000* epizód pre algoritmus *One-Step SARSA*. V nasledujúcom grafe sú zobrazené priebehy učenia aj pre zvyšné modifikácie *Eligibility Traces SARSA* a *Q-SARSA(λ)*.



Obr. 8.6: Graf SARSA učenia v prostredí Lunar Landing

Z grafu 8.6 je viditeľné, že všetky typy algoritmov boli pomerne nestabilné pri riešení tohto prostredia. Algoritmus *One-step SARSA* konvergoval po najmenšom počte epizód a všeobecne dosahoval najväčšej stability. Modifikované algoritmy *Eligibility traces SARSA* a *Q-SARSA(λ)* dosahovali nižšej stability ako aj odmeny. Vzhľadom k využívaniu modifikácie *Eligibility traces* v spomínaných modifikovaných algoritmoch učenie, respektíve vykonanie 15000 epizód trvalo o dosť dlhší čas ako pre základný algoritmus *One-step SARSA*.

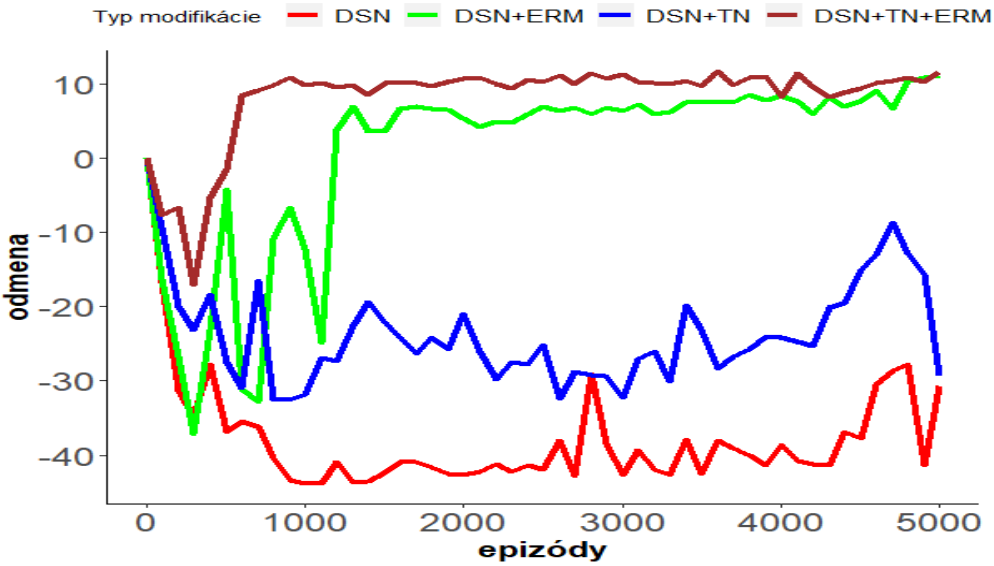
8.1.2.2 Deep SARSA učenie

Ďalšími experimentami tak ako aj v prostredí *Cart Pole balancing* boli experimenty s Deep SARSA učením a jeho modifikáciami. Vzhľadom ku komplexnosti prostredia bolo potrebné upraviť pôvodnú konfiguráciu pre učenie. Bolo potrebné predĺžiť dobu explorácie a taktiež znížiť hyperparameter *Learning rate*, zvýšenie počtu skrytých vrstiev sa taktiež ukázalo ako vhodná voľba.

Update optimizer	Adam
Learning Rate	0.0002
Gamma	0.99
Počet skrytých vrstiev	2
Počet neurónov v skrytej vrstve	256
Epsilon start	1.0
Epsilon min	0.05
Epsilon decay	0.99995

Tabuľka 8.4: Parametry Deep SARSA učenia pre prostredie Lunar Landing

Z grafu 8.7 je viditeľné, že samotný algoritmus Deep SARSA network nebol schopný skonvergovať do správneho riešenia, taktiež sa mu to nepodarilo s využitím modifikácie *Target Network*. Ako sa ukázalo aj v experimentoch v prostredí *Cart Pole balancing*, významnú úlohu pre správnu konvergenciu zastupovala modifikácia *Experience memory replay*, vďaka ktorej bol algoritmus schopný konvergovať ku riešeniu. Využitím oboch modifikácií sa dovŕšila stabilita a rýchlejšia konvergencia.



Obr. 8.7: Graf efektivity modifikácií Deep SARSA network

8.2 Paralélny beh

V tejto sekcii sa nachádzajú experimenty vykonané pomocou paralelného učenia s architektúrou *Gorila* a *Bundle*. Tieto experimenty boli zamerané na otestovanie ako dátový paralelizmus dokáže ovplyvniť rýchlosť učenia a úspešnosť riešenia prostredia za využitia Deep Sarsa učenia. Cieľom experimentov bolo taktiež odhaliť vplyv sieťovej komunikácie na efektivitu učenia, keďže superpočítač ponúkal možnosť využiť viacero nezávislých uzlov. Akonáhle bola implementácia dátového paralelizmu funkčná na lokálnom prostredí, bolo možné využiť túto implementáciu v prostredí superpočítača.

Využitý superpočítač bol superpočítač *Barbora* od spoločnosti *IT4Innovations* [16]. Implementácia bola konkrétne spúšťaná na uzloch bez akcelerátorov, kde týchto uzlov bolo presnejšie 192. Tieto uzly komunikovali medzi sebou za pomoci technológie *Mellanox InfiniBand*.

Konfigurácia uzlu superpočítača Barbora	
OS	Linux Server release 7.9 (Maipo)
CPU	2x Intel Cascade Lake 6240, 18-core, 2.6 GHz
RAM	192 GB DDR (12x 16 GB)

Tabuľka 8.5: Konfigurácia uzlu superpočítača Barbora

Prostredie superpočítača vyžadovalo spúšťanie programu pod konkrétnou dávkou, ktorá bola definovaná typom fronty a počtom uzlov počítača.

8.2.1 Cart Pole balancing

Toto prostredie som zvolil pre otestovanie paralelného učenia vzhľadom k jeho jednoduchosti a pomernému rýchlemu dosiahnutiu správneho riešenia. Všetky experimenty a ich efektivita sú popísané v čase na rozdiel od sekvenčného učenia, kde sa využívali epizódy. Čas sa využíva na základe otestovania rýchlosti využitia dátového paralelizmu. Všetky experimenty v paralélnom behu trvali maximálne 600 sekúnd, t.j. 10 minút.

8.2.1.1 Architektúra Gorila

Prvými experimentami v paralélnom behu boli experimenty s využitím implementácie architektúry *Gorila*. Ako je uvedené v kapitole 6.4 o implementácii tejto architektúry existujú všelijaké možnosti využitia počtu komponent *Learner* a *Actor*. Ja som sa rozhodol využiť spomínaný *Bundled* mód. V tomto móde sa používa taký istý počet komponent *Actor* ako je zvolený počet komponent *Learner*.

Pre učenie som využil nasledovnú konfiguráciu:

V prvej sade experimentov som menil počet dávky skúseností, ktoré boli využívané pre výpočet gradient. Kde tento gradient bol následne zaslaný na komponentu *Parameter server*.

Konfigurácia paralelného učenia s architektúrou Gorila	
Update optimizer	Adam
Learning Rate	0.00001
Gamma	0.99
Počet skrytých vrstiev	2
Počet neurónov v skrytej vrstve	256
Epsilon start	1.0
Epsilon min	0.05
Epsilon decay	0.999995
Veľkosť dávky skúseností pre Gradient	256
Typ aplikácie gradientov	Average
Čakať na gradienty od zvyšných komponent	true

Tabuľka 8.6: Konfigurácia paralelného učenia s architektúrou Gorila

Veľkosť dávky	**Čas v sekundách**	**Dosiahnutá odmena**	**Prostredie vyriešené**
16	600	56	nie
32	600	62	nie
64	600	83	nie
128	550	197	áno
256	455	199	áno
512	490	196	áno

Tabuľka 8.7: Vplyv veľkosti dávky skúseností na učenie v architektúre Gorila

Experimenty z tabuľky 8.7 ukázali, že nízky počet dávky skúseností nie je vhodný pre učenie. Najvhodnejšou hodnotou pre dávku bolo číslo 256, následné zvyšovanie tejto hodnoty sa taktiež ukázalo ako nevhodné vzhľadom k tomu, že riešenie prostredia bolo dosiahnuté za dlhší čas. Tieto experimenty využívali *Bundled* mód s hodnotou $N_{act\&lea} = 4$.

V druhej sade experimentov som sa rozhodol využívať iba jeden distribuovaný uzol. Na tomto uzle som postupne pridával počet komponent *Learner* a *Actor*. Keďže jeden uzol superpočítača obsahuje až 32 jadier, mohol som využiť všetky tieto jadrá a vyskúšať *Bundled* mód s počtom 8. Začal som na hodnote $N_{act\&lea} = 1$. Túto hodnotu som vždy zvyšoval násobkom čísla 2.

Počet inštancií $N_{act\&lea}$	**Čas v sekundách**	**Prostredie vyriešené**
1	533	áno
2	485	áno
4	455	áno
8	575	áno

Tabuľka 8.8: Tabuľka výsledkov experimentov nad jedným uzlom pre architektúru Gorila

Z tabuľky 8.8 je viditeľné, že všetky experimenty boli schopné dosiahnuť riešenia prostredia. Jednotlivé experimenty boli vykonávané vždy tri krát, následne bol vypočítaný priemerný čas, tento proces bol vykonávaný kvôli tomu, že priebeh učenia bol pomerne nestabilný a často trval inú časovú dobu. Taktiež je badateľné, že najvhodnejšou hodnotou sa ukázala byť $N_{act\&lea} = 4$.

Paralélny beh pomocou architektúry Gorila sa ukázal byť celkom nestabilný. Táto nestabilita bola pravdepodobne spôsobená asynchrónnym prístupom pre prijímanie parametrov, skúseností a gradient. Vykonal som rôzne úpravy ako zmenu rôznych hyperparametrov, úpravu implementácie pre čiastočne synchrónny prístup. Avšak učenie bol aj tak naďalej nestabilné, dosiahnutá odmena často rapídne klesala až na bod počiatku učenia. Tento neúspech mohol byť zapríčinený chybou implementácie, avšak túto možnú chybu sa mi nepodarilo nájsť.

Vzhľadom k tomuto neúspechu som sa rozhodol nepokračovať ďalej v experimentoch na viacerých distribuovaných uzloch pre túto architektúru. Ako sa ukázalo už počas implementácie oboch architektúr, nasledujúca architektúra *Bundle* riešila problém nestability pomocou úplného synchrónneho prístupu. Nasledujúce experimenty sa teda zaoberajú touto architektúrou.

8.2.1.2 Architektúra Bundle

Vzhľadom k nestabilite učenia predošlej architektúry som využil pre učenie architektúru *Bundle*, ktorá znížila komunikáciu medzi komponentami. Týmto krokom som sa snažil dosiahnuť zrýchlenie učenia a taktiež zvýšenie stability. Narozdiel od *Gorila* architektúry, táto architektúra fungovala synchrónne a to tak, že vždy pre zadaný počet epizód všetky *Bundle* komponenty zaslali svoje gradienty na *Parameter server*. Taktiež všetky v tom istom čase prijímali parametre siete θ^+.

V prvej sade experimentov som menil veľkosť dávky skúseností vďaka ktorej sa vypočítaval gradient, ktorý bol zasielaný na komponentu *Parameter server*. Počet inštancií komponent bundle bol pevne stanovený na 4.

Z tabuľky 8.9 je viditeľné, že najvhodnejším počtom pre dávku bolo číslo 256. Vyšší počet už začínal spomaľovať učenie, kde narozdiel od výrazne menšieho počtu algoritmus nebol schopný vyriešiť prostredie.

Veľkosť dávky	**Čas v sekundách**	**Dosiahnutá odmena**	**Prostredie vyriešené**
16	600	134	nie
32	600	176	nie
64	560	196	áno
128	210	198	áno
256	208	196	áno
512	221	196	áno

Tabuľka 8.9: Vplyv veľkosti dávky skúseností na učenie

V ďalšej sade experimentov pre túto architektúru som obdobne ako v predošlej sekcii spúšťal učenie iba na jednom distribuovanom uzle. Pre učenie som využil nasledovnú konfiguráciu:

Konfigurácia paralelného učenia s architektúrou Bundle	
Update optimizer	Adam
Learning Rate	0.0006
Gamma	0.99
Počet skrytých vrstiev	2
Počet neurónov v skrytej vrstve	256
Epsilon start	1.0
Epsilon min	0.05
Epsilon decay	0.999995
Veľkosť dávky skúseností pre Gradient	256
Typ aplikácie gradientov	Average
Čakať na gradienty od zvyšných komponent	true

Tabuľka 8.10: Konfigurácia paralelného učenia s architektúrou Bundle

V tabuľke 8.11 sú zobrazené výsledky experimentov nad jedným uzlom. Z tejto sady experimentov bolo zistené, že najvhodnejším počtom inštancií pre paralélny beh na jednom distribuovanom uzle je číslo 4. Do tohto počtu sa učenie zrýchľovalo. Ďalším zvýšením počtu inštancií sa učenie, respektíve dosiahnutie vyriešenia prostredia spomaľovalo. Využitím počtu až 16 inštancií dokonca toto učenie nedosiahlo požadovanú odmenu pre vyriešenie prostredia. Tento neúspech bol pravdepodobne zapríčinený pretrénovaním siete komponenty *Parameter server* vysokým počtom čiastočne tých istých parametrov z rôznych inštancií. Zmeny hyperparametrov ako *Learning rate, Epsilon decay, Batch size* nepriniesli potrebnú zmenu, ktorá by pomohla dosiahnuť riešenia prostredia.

Počet inštancií	**Čas v sekundách**	**Priemerná odmena**	**Prostredie vyriešené**
1	743	102	áno
2	310	98	áno
4	233	81	áno
8	448	115	áno
16	600	85	nie

Tabuľka 8.11: Tabuľka výsledkov experimentov nad jedným uzlom pre architektúru Bundle

V nasledujúcej sade experimentov som menil počet využívaných distribuovaných uzlov pre počet inštancií 4 na ktorých bolo spúšťané učenie. Pre porovnanie som využil výsledky z predošlej sady experimentov, ktoré reprezentovali učenie na jednom distribuovanom uzle.

Počet uzlov	Počet inštancií	Čas v sekundách	Priemerná odmena	Prostredie vyriešené
1	4	233	81	áno
2	2	226	85	áno
4	1	205	98	áno

Tabuľka 8.12: Tabuľka výsledkov experimentov nad viacerými uzlami pre architektúru Bundle

Z nasledujúcej tabuľky 8.12 vyplýva, že využitie viacerých uzlov pre paralélny beh čiastočne urýchľuje proces učenia, teda dosiahnutia požadovaného výsledku v danom prostredí. Proces učenia na jednom uzle je teda pravdepodobne spomaľovaný počtom inštancií, ktoré sú na ňom spúšťané.

Vzhľadom k tomuto poznatku som skúsil vykonať podobnú sadu experimentov pre celkový počet komponent 16. Z tabuľky 8.13 je teda viditeľné, že využitie viacerých distribuovaných uzlov pre ten istý celkový počet komponent prispieva ku rýchlejšiemu učeniu.

Počet uzlov	Počet inštancií	Čas v sekundách	Priemerná odmena	Prostredie vyriešené
1	16	600	85	nie
2	8	546	98	áno
4	4	287	88	áno

Tabuľka 8.13: Tabuľka výsledkov experimentov nad viacerými uzlami pre architektúru Bundle

Kapitola 9

Záver

Oblasť deep reinforcement learning s využitím algoritmu Deep SARSA network ponúka širokú škálu využitia v rôznych odvetviach. Tento algoritmus ukázal, že je schopný konkurovať algoritmu DQN aj keď často dosahuje o niečo nižších výsledkov.

V tejto práci som popisoval riešenie kombinácie deep learning siete a reinforcement learning algoritmu SARSA. Toto riešenie som integroval ako samostatný modul do aplikácie *NeuronNet-Modeler*. V tomto module bolo možné učiť algoritmus na rôznych prostrediach a prostredníctvom rôznych modifikácií. Samotný algoritmus SARSA bol schopný riešiť dané testovacie prostredie aj keď vyžadoval normalizáciu stavov a jeho učenie bolo mierne nestabilné. Algoritmus Deep SARSA network využíval umelú neurónovú sieť pre zvýšenie stability a efektivity.

Nevýhodou tohto algoritmu bolo dlhšie konfigurovanie hyperparametrov pre konkrétny problém, keďže každé trénovacie prostredie predstavovalo iný level zložitosti. Taktiež dlhší čas pre naučenie a vysoké využitie zdrojov hardvéru. Táto potreba pre vysoký výpočetný výkon sa dala vyriešiť pomocou paralélneho behu na viacerých strojoch, ktoré dokázali medzi sebou komunikovať cez sieť.

Toto riešenie som využil pre paralelizáciu učenia algoritmu Deep SARSA network, kde som najprv využil architektúru Gorila. Spomínaná architektúra fungovala avšak s vysokou dávkou nestability. Na základe tohto problému som využil pre paralélne učenie ďalšiu architektúru, ktorá bola schopná dovŕšiť kvalitnejších výsledkov v prostredí Cart pole balancing.

Ako posledné som vykonal experimenty ako pre sekvenčný beh tak aj paralélny beh algoritmu. V časti, popisujúcej sekvenčný beh som otestoval funkčnosť implementácie jednotlivých algoritmov a zistil ktoré modifikácie boli najefektívnejšie. Pre časť paralélneho behu som otestoval vplyv dátovej paralelizácie na algoritmus Deep SARSA network. V jednotlivých sadách experimentov som zistil najvhodnejšiu veľkosť dávky skúseností pre výpočet gradient. Taktiež som zistil najefektívnejší počet komponent, kedy sa učenie stále zrýchľovalo.

V budúcnosti by bolo možné otestovať vplyv ďalších existujúcich modifikácií pre algoritmy deep reinforcement learning. Taktiež by stálo za pokus zistiť efektivitu paralelizácie bez sieťovej komunikácie pomocou vlákien na jednom zariadení.

Literatúra

1. *What is Machine Learning? A Definition.* [Online]. 2020 [cit. 2020-06-05]. Dostupné z: https://www.expert.ai/blog/machine-learning-definition/.
2. BELLMAN, Richard. A Markovian Decision Process. *Indiana Univ. Math. J.* 1957, roč. 6, s. 679–684. ISSN 0022-2518.
3. JONES, Morgan; PEET, Matthew M. A generalization of Bellman's equation with application to path planning, obstacle avoidance and invariant set estimation. *Automatica*. 2021, roč. 127, s. 109510. ISSN 0005-1098. Dostupné z DOI: https://doi.org/10.1016/j.automatica.2021.109510.
4. SUTTON, Richard S.; BARTO, Andrew G. *Reinforcement Learning: An Introduction*. Cambridge, Massachusetts, London, England: The MIT Press, 2014.
5. SCHMIDHUBER, Jürgen. Deep Learning in Neural Networks: An Overview. *CoRR*. 2014, roč. abs/1404.7828. Dostupné z arXiv: 1404.7828.
6. SCHMIDHUBER, Jürgen. Deep learning in neural networks: An overview. *Neural Networks*. 2015, roč. 61, s. 85–117. ISSN 0893-6080. Dostupné z DOI: https://doi.org/10.1016/j.neunet.2014.09.003.
7. BELLEMARE, Vincent Francois-Lavet; Peter Henderson; Riashat Isla ; Marc G.; PINEAU, Joelle. An Introduction to Deep Reinforcement Learning. *Foundations and Trends in Machine Learning*. 2018, roč. 11, s. 1–14. Dostupné z arXiv: arXiv:1507.04296v2.
8. *What are recurrent neural networks?* [Online]. 2021 [cit. 2020-09-14]. Dostupné z: https://www.ibm.com/cloud/learn/recurrent-neural-networks.
9. *Deep Belief Networks — all you need to know* [online]. 2018 [cit. 2018-08-01]. Dostupné z: https://medium.com/@icecreamlabs/deep-belief-networks-all-you-need-to-know-68aa9a71cc53.
10. *A Hands-On Introduction to Deep Q-Learning using OpenAI Gym in Python: Deep Q-Learning* [online]. 2019 [cit. 2019-04-18]. Dostupné z: https://www.analyticsvidhya.com/blog/2019/04/introduction-deep-q-learning-python/.

11. *RL — DQN Deep Q-network: RL — DQN Deep Q-network* [online]. 2018 [cit. 2018-07-16]. Dostupné z: `https://jonathan-hui.medium.com/rl-dqn-deep-q-network-e207751f7ae4`.

12. *Deep Learning for Java: Deep Learning for Java* [online]. 2020 [cit. 2020-05-14]. Dostupné z: `https://deeplearning4j.org/`.

13. *OpenAI gym: Deep Learning for Java* [online]. 2021 [cit. 2021-03-14]. Dostupné z: `https://gym.openai.com/`.

14. GOOGLE DEEPMIND, London. Massively Parallel Methods for Deep Reinforcement Learning. *Massively Parallel Methods for Deep Reinforcement Learning*. 2015, roč. 11, s. 1–14. Dostupné z arXiv: `arXiv:1507.04296v2`.

15. *Aeron: Aeron Library* [online]. 2021 [cit. 2021-02-14]. Dostupné z: `https://github.com/real-logic/aeron`.

16. *It4Innovations: It4Innovations website* [online]. 2021 [cit. 2020-03-14]. Dostupné z: `https://www.it4i.cz/`.

Dodatok A

Popis prostredí

A.1 Cart Pole balancing

V tomto prostredí agent algoritmu pohyboval vozíkom pomocou dvoch možných akcií: vľavo a vpravo. Účelom využitia týchto akcií bolo udržanie žrde v kolmej polohe voči vozíku. Toto prostredie sa označovalo za vyriešené, akonáhle priemerný počet krokov za epizódu dosiahol počtu vyššieho ako 195. Stav prostredia sa skladal zo 4 atribútov:

- poloha vozíka
- rýchlosť vozíka
- uhol žrde
- rýchlosť žrde na špičke

Toto prostredie existovalo v niekoľkých podobách, kde niektoré verzie nechávajú žrď krúžiť po svojej osy, iné verzie zasa celé prostredie obnovujú do počiatočného stavu po prekročení istého uhlu žrde. Implementácia môjho prostredia využívala druhú možnosť.

V nasledujúcej tabuľke sa nachádza popis jednotlivých atribútov stavu.

Atribút	**Minimálna hodnota**	**Maximálna hodnota**
Poloha vozíka	-2.0	2.0
Rýchlosť vozíka	$-\infty$	∞
Uhol žrde	-12º	12º
Rýchlosť žrde na špičke	$-\infty$	∞

Tabuľka A.1: Rozsah jednotlivých atribútov stavu v Cart Pole balancing

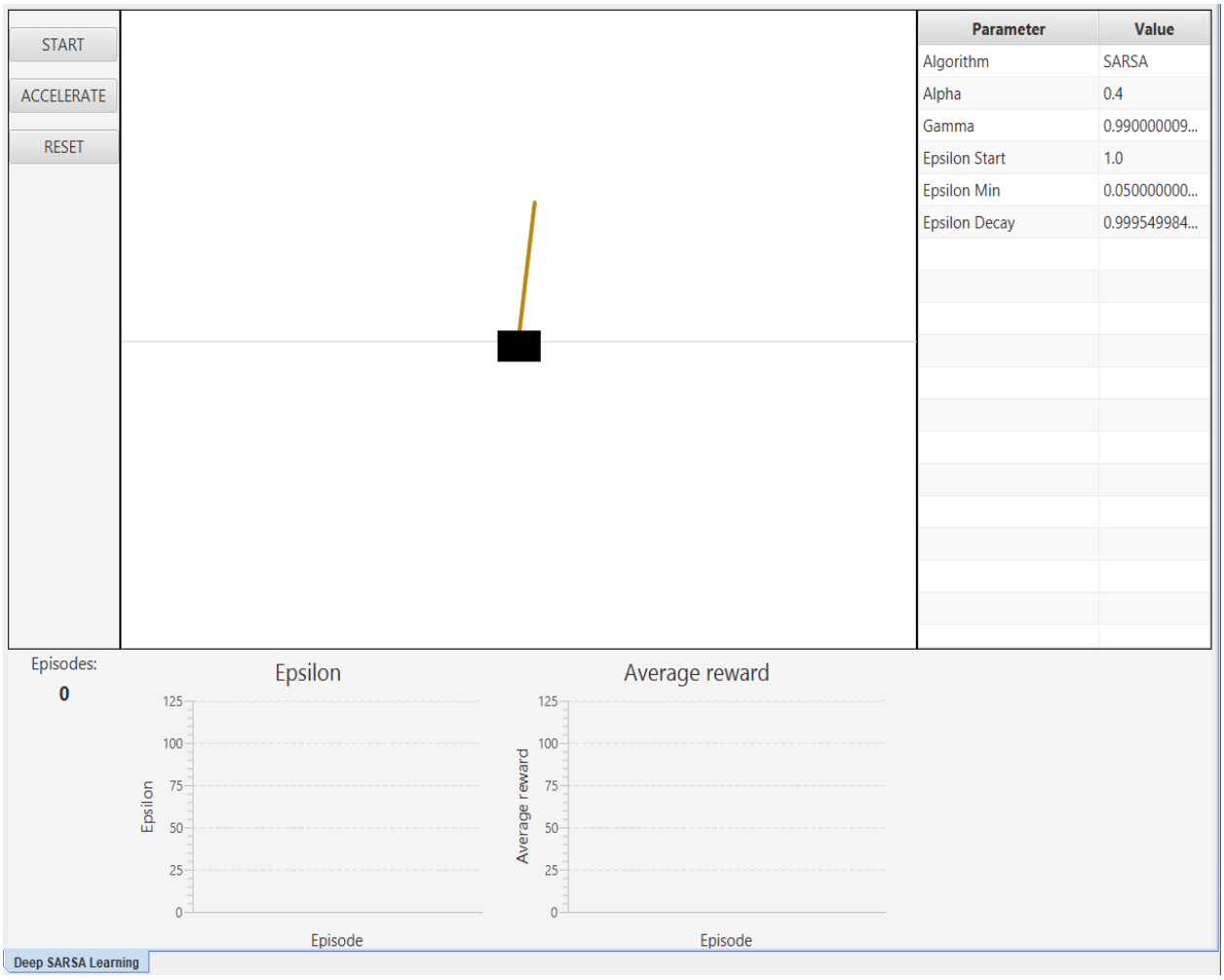
Pre získanie uhlu žrde a rýchlosti žrde na špičke som využil nasledovné fyzikálne vzorce:

- $\Phi t + 1 = \Phi t + \Phi_{t+1} \Delta t$
- $\Phi_{t+1} = \frac{\Phi_t + g_{sin}(\Phi_t) - \alpha \cdot m \cdot d(\Phi_t)^2 sin(2\Phi_t)/2 + \alpha \cdot cos(\Phi_t) a_t}{4l/3 - \alpha \cdot m \cdot d \cdot cos^2(\Phi_t) \Delta t} \cdot \Delta_t$

Na základe týchto rozsahov atribútov som taktiež zvolil ukončovacie podmienky:

- Presiahnutie hodnoty uhlu žrde v akomkoľvek smere hodnotu 12, respektíve -12
- Maximálna dĺžka epizódy je 200 krokov
- Vozík presiahne svojim bokom jednu z hraničných stien prostredia

Odmena pre agenta za akýkoľvek stav, vrátane konečného stavu bola stanovená na hodnotu 1.



Obr. A.1: Ukážka prostredia Cart Pole balancing v projekte NeuronNetModeler

A.2 Lunar Landing

Toto prostredie popisovalo o niečo komplexnejší problém v porovnaní s prostredím *Cart Pole balancing*. Úlohou agenta bolo bezpečne pristať pomocou rakety na pristávaciu platformu, ktorá bola na povrchu Mesiaca. Agent mal možnosť vykonávať vždy jednu zo štyroch možných akcií: použiť pravý motor(vľavo), použiť ľavý motor(vpravo), žiadna akcia, použitie hlavného motora(hore).

Prostredie sa považovalo za vyriešené akonáhle agent dokázal bezpečne pristáť na platformu v percentuálnej úspešnosti 85 percent za 100 epizód. Stav prostredia sa skladal z nasledovných atribútov:

- Horizontálna poloha rakety
- Vertikálna poloha rakety
- Horizontálna rýchlosť rakety
- Vertikálna rýchlosť rakety
- Náklon(uhol) rakety
- Zostatok paliva hlavného motora rakety

Pre urýchlenie učenia boli využívané dodatočné, tzv. ideálne atribúty, ktoré popisovali najvhodnejšie hodnoty pre konkrétny atribút. Tieto atribúty boli nasledovné:

- Ideálna horizontálna poloha rakety
- Ideálna vertikálna poloha rakety
- Ideálna vertikálna rýchlosť rakety

Pre výpočet vertikálnej rýchlosti bol využitý nasledovný fyzikálny vzorec:

$$h = h_0 + v_0 \Delta t + a(\Delta t)^2/2$$

Kde jednotlivé prvky znamenajú:

- h_0 vertikálna pozícia v predošlom časovom kroku
- v_0 rýchlosť v predošlom časovom kroku
- a - akcelerácia rakety, v prípade využitia hlavného motora platí, že $a = A - g$, inak $a = -g$
- Δt - malý časový krok

V nasledujúcej tabuľke sa nachádza popis jednotlivých atribútov stavu.

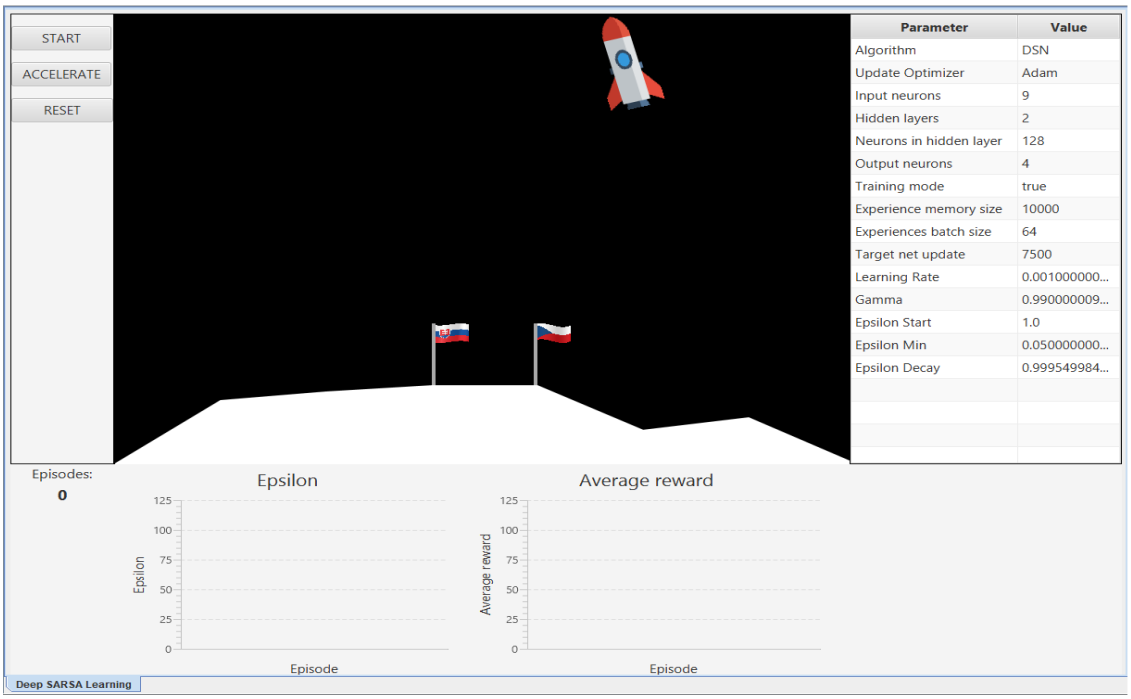
Atribút	Minimálna hodnota	Maximálna hodnota	Ideálna hodnota
Vertikálna poloha rakety	0	430	*poloha platformy*
Horizontálna poloha rakety	0	800	*poloha platformy*
Vertikálna rýchlosť rakety	$-\infty$	∞	*<0;-3>*
Horizontálna rýchlosť rakety	-3	3	*0*
Náklon rakety	-35	35	*0*
Palivo rakety	0	700	*700*

Tabuľka A.2: Rozsah jednotlivých atribútov stavu v Lunar landing

Na základe týchto jednotlivých rozsahov som taktiež určil ukončovacie podmienky:

- Presah počtu maximálnych krokov 1500
- Presah náklonu rakety v jednom zo smerov
- Presah horizontálnych stien
- Dotyk s povrchom Mesiaca

Odmena pre agenta bola vždy vypočítavaná zo všetkých spomínaných atribútov stavu. Pre správne pristátie dostal agent dodatočnú pozitívnu odmenu, narozdiel pre zrážku rakety s povrchom alebo stenou, bola odmena znížená o ďalšiu hodnotu.



Obr. A.2: Ukážka prostredia Lunar landing v projekte NeuronNetModeler